第3章 第一个样例程序及工程组织

本章阐述“入门”过程的（5）—（8）步，通过这个过程，完成第一个 CodeWarrior 工程、IAR 工程的入门。利用 GPIO 模块编程控制发光二极管作为入门例子，给出 CodeWarrior、IAR 工程组织、框架，阐述各个文件的功能，主要目的是使读者理解程序框架和工作过程。重点是透彻理解第一工程的执行过程。

3.1 通用I/O接口基本概念及连接方法

1．I/O 接口的概念

I/O 接口，即输入输出接口，是微控制器同外界进行交互的重要通道。这里的接口英文是 port，也可以翻译为“端口”，另一个英文单词是 interface，也翻译为接口。从中文字面看，接口与端口似乎有点区别，但在嵌入式系统中它们的含义是相同的。有时 I/O 引脚称为接口（interface），而把用于对 I/O 引脚进行编程的寄存器称为端口（port），实际上它们是紧密相连的。因此，不必深究它们之间的区别。有些书中甚至直接称 I/O 接口（端口）为 I/O 口。在嵌入式系统中，接口千变万化，种类繁多，有显而易见的人机交互接口，如操纵杆、键盘、显示器；也有无人介入的接口，如网络接口、机器设备接口。

2．通用 I/O

所谓通用 I/O，也记为 GPIO（General Purpose I/O），即基本的输入/输出，有时也称并行 I/O，或普通 I/O，它是 I/O 的最基本形式。本书中使用正逻辑，电源（Vcc）代表高电平，对应数字信号“1”；地（GND）代表低电平，对应数字信号“0”。作为通用输入引脚，MCU 内部程序可以通过端口寄存器读取该引脚，知道该引脚是“1”（高电平）或“0”（低电平），即开关量输入。作为通用输出引脚，MCU 内部程序通过端口寄存器向该引脚输出“1”（高电平）或“0”（低电平），即开关量输出。大多数通用 I/O 引脚可以通过编程来设定工作方式为输入或输出，称之为双向通用 I/O。

3．上拉下拉电阻与输入引脚的基本接法

芯片输入引脚的外部有三种不同的连接方式：带上拉电阻的连接、带下拉电阻的连接和“悬空”连接。通俗地说，若 MCU 的某个引脚通过一个电阻接到电源（Vcc）上，这个电阻被称为“上拉电阻”。与之相对应，若 MCU 的某个引脚通过一个电阻接到地（GND）上，则相应的电阻被称为“下拉电阻”。这种做法使得，悬空的芯片引脚被上拉电阻或下拉电阻初始化为高电平或低电平。根据实际情况，上拉电阻与下拉电阻可以取值在 1KΩ～10KΩ之间，其阻值大小与静态电流及系统功耗相关。

图 3-1 给出了一个 MCU 的输入引脚的三种外部连接方式，假设 MCU 内部没有上拉或下拉电阻，图中的引脚 I3 上的开关 K3 采用悬空方式连接就不合适，因为 K3 断开时，引脚 I3 的电平不确定。在图 3-1 中，R1>>R2，R3<<R4，各电阻的典型取值为：R1=20K，R2=1K，R3=10K，R4=200K。

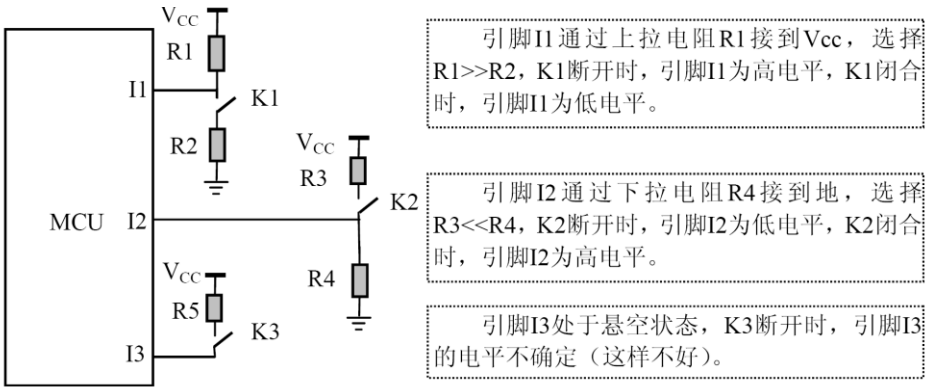

图 3-1 I/O 口输入电路

4．输出引脚的基本接法

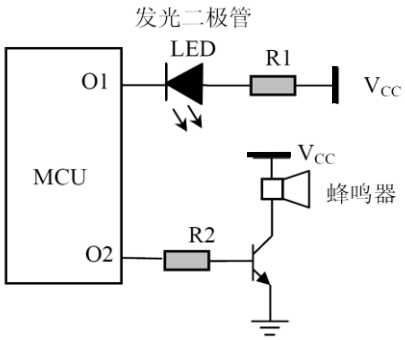

图 3-2 I/O 口输出电路

作为通用输出引脚，MCU 内部程序向该引脚输出高电平或低电平来驱动器件工作，即开关量输出。如图 3-2 所示。

一种接法是 O1 引脚直接驱动发光二极管 LED，当 O1 引脚输出高电平时，LED 不亮；当 O1 引脚输出低电平时，LED 点亮。这种接法的驱动电流一般在 2mA～10mA。

另种接法是 O2 引脚通过一个 NPN 三极管驱动蜂鸣器，当 O2 脚输出高电平时，蜂鸣器响；O2 脚输出低电平时，蜂鸣器不响。这种接法的驱动电流可达 100mA 左右，而 O2 引脚控制电流可以在几个 mA 左右。

若负载需要更大的驱动电流，就必须另外的驱动电路，但对 MCU 编程来说，没有任何影响。

3.2 MK60N512VMD100的GPIO

MK60N512VMD100 的大部分引脚具有多重功能，可以通过编程设定使用其中一种功能。GPIO 作为基本功能，MK60N512VMD100 有 5 个 GPIO 口，共 100 个引脚。分别为 A 口、B 口、C 口、D 口、E 口。A 口有 26 个引脚，分别为 PTA0~PTA19、PTA24~PTA29，B 口有 20 个引脚，分别为 PTB0~PTB11、PTB16~PTB23，C 口有 20 引脚，分别为 PTC0~PTC19，D 口有 16 个引脚，分别为 PTD0~PTD15，E 口有 18 个引脚，分别为 PTE0~PTE12、PTE24~PTE28。

每个口均包含 6 个寄存器，下面分别介绍一下。

1．数据输出寄存器（GPIOx_PDOR），可读写，32 位，复位时为 0。无论引脚上的逻辑电平为 0 还是为 1，相应的引脚都被配置为输出。

2．数据输入寄存器（GPIOx_PDIR），只读，32 位，复位时为 0。读为 0 时，说明相应引脚上为低电平或配置为数字功能，读为 1 时，说明相应引脚上为高电平。

3．数据方向寄存器（GPIOx_PDDR），可读写，32 位，复位时为 0。0——定义为输入，1——定义为输出。

4．输出设置寄存器（GPIOx_PSOR），可写，32 位，复位时为 0。对该寄存器进行写操作将该表 PDOR 寄存器的值。写 0 时，不改变 PDOR 上的相应位，写 1 时，将 PDOR 上的

相应位置 1。

5．输出清除寄存器（GPIOx_ PCOR），可写，32 位，复位时为 0。对该寄存器进行写操作将该表 PDOR 寄存器的值。写 0 时，不改变 PDOR 上的相应位，写 1 时，将 PDOR 上的相应位清 0。

6．输出触发寄存器（GPIOx _PTOR），可写，32 位，复位时为 0。对该寄存器进行写操作将该表 PDOR 寄存器的值。写 0 时，不改变 PDOR 上的相应位，写 1 时，将 PDOR 上的相应位反转。

GPIO 的基本编程方法：

（1）通过“数据方向寄存器”设置相应引脚为输入或输出；

（2）若是输出引脚，则通过“数据输出寄存器”设置引脚输出高电平或低电平；

（3）若是输入引脚，则通过“数据输入寄存器”获得引脚的状态。

（4）若是输出引脚，可通过“输出设置寄存器”、“输出清除寄存器”、“输出触发寄存器”来改变引脚的状态。

3.3 开发环境与JTAG写入器

嵌入式软件开发有别于桌面软件开发的一个显著的特点，是它一般需要一个交叉编译和调试环境，即编辑和编译软件在通常的 PC 机上进行，而编译好的软件需要通过写入工具下载到目标机上执行，如 MK60N512VMD100 的目标机上。由于主机和目标机处理器的体系结构彼此不同，从而增加了嵌入式软件开发的难度。所以选择一些好的开发套件有助于对目标机的学习与开发。本书将介绍 IAR Systems 公司的 IAR Embedded Workbench for ARM 6.10 集成开发环境（简称 IAR 环境）、Freescale 公司的 CodeWarrior10.1 集成开发环境（简称 CW 环境）、苏州大学的 MK60N512VMD100 硬件评估板以及 JTAG 写入器。

3.3.1 IAR 开发环境简介与基本使用方法

1．IAR 环境功能和特点

Embedded Workbench for ARM 6.10 是 IAR Systems 公司为 ARM 微处理器开发的一个集成开发环境。比较其他的 ARM 开发环境，IAR 具有入门容易、使用方便和代码紧凑等特点。

IAR 中包含一个全软件的模拟程序(simulator)。用户不需要任何硬件支持就可以模拟各种 ARM 内核、外部设备甚至中断的软件运行环境。从中可以了解和评估 IAR 的功能和使用方法。

IAR 的主要特点如下：

1、高度优化的 IAR ARM C/C++ Compiler

2、IAR ARM Assembler

3、一个通用的 IAR XLINK Linker

4、IAR XAR 和 XLIB 建库程序和 IAR DLIB C/C++运行库

5、功能强大的编辑器

6、项目管理器

7、命令行实用程序

8、IAR C-SPY 调试器(先进的高级语言调试器)

2．IAR 环境安装与设置

IAR 环境安装并不复杂，按提示步骤一步步来即可，光盘中给出了详细的安装步骤。关键要注意 IAR 环境的设置，这部分光盘中也给出了操作步骤。IAR 环境的运行界面如图 3-3 所示。

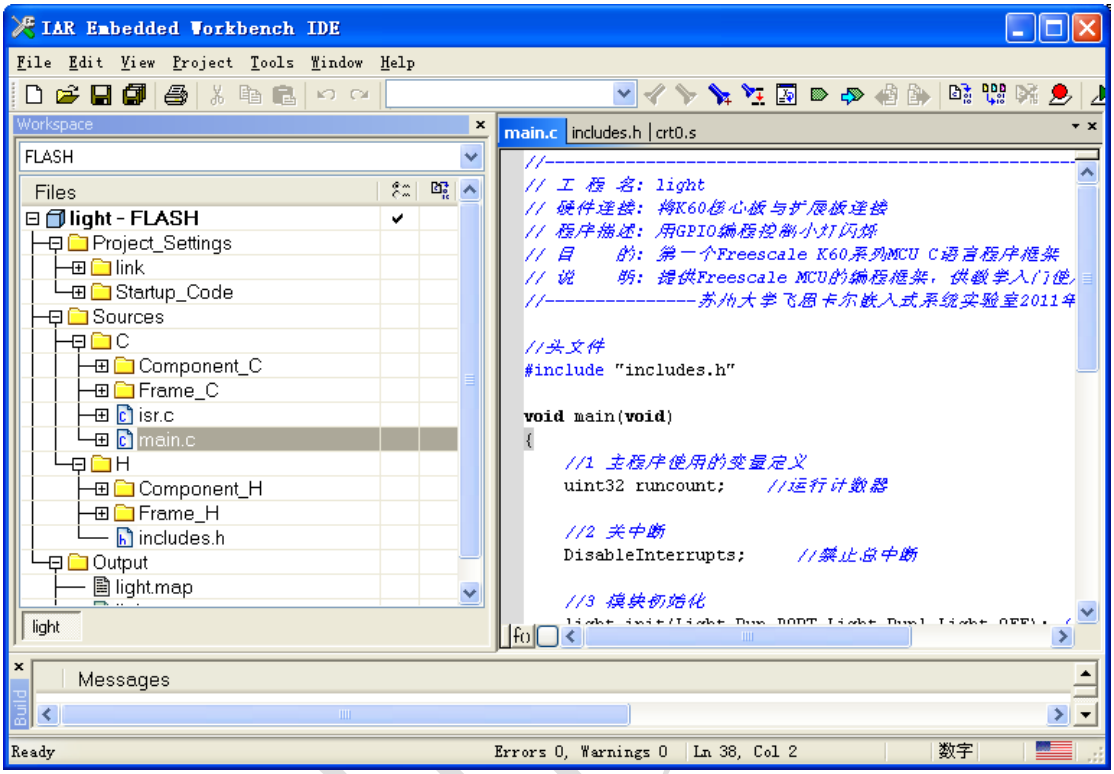

图 3-3 IAR 环境的运行界面

3.3.2 CW 开发环境简介与基本使用方法

1．CW 环境功能和特点

CodeWarrior 开发环境(简称 CW 环境)是 Freescale 公司研发的面向 Freescale MCU 与 DSP 嵌入式应用开发的商业软件工具，其功能强大，是 Freescale 向用户推荐的产品。

CodeWarrior 分为 3 个版本：特别版（Special Edition）、标准版和专业版。特别版是免费的，用于教学目的，对生成的代码量有一定限制，C 语言代码不得超过 12KB，对工程包含的文件数目也限制在 30 个以内。标准版和专业版没有这种限制。3 个版本的区别在于用户所获取的授权文件（license）不同，特别版的授权文件随安装软件附带，不需要特殊申请，标准版和专业版的授权文件需要付费。CodeWarrior 特别版、标准版和专业版的定义随所支持的微处理器的不同而不同，如 CodeWarrior for HC08 V6.0、CodeWarrior for HC12 V4.6、CodeWarrior for ColdFire V6.3 等,本书使用 CodeWarrior V10.1，这个版本支持所有系列的芯片。

CW 环境包括以下几个功能模块：编辑器、源码浏览器、搜索引擎、构造系统、调试器、工程管理器。编辑器、编译器、连接器和调试器对应开发过程的四个主要阶段，其它模块用以支持代码浏览和构造控制，工程管理器控制整个过程。该集成环境是一个多线程应用，能在内存中保存状态信息、符号表和对象代码，从而提高操作速度；能跟踪源码变化，进行自

动编译和连接。

2．CW 环境安装与设置

CW 环境安装没有什么特别之处，在 Windows 操作系统上，只要按照安装向导单击鼠标就可以自动完成。

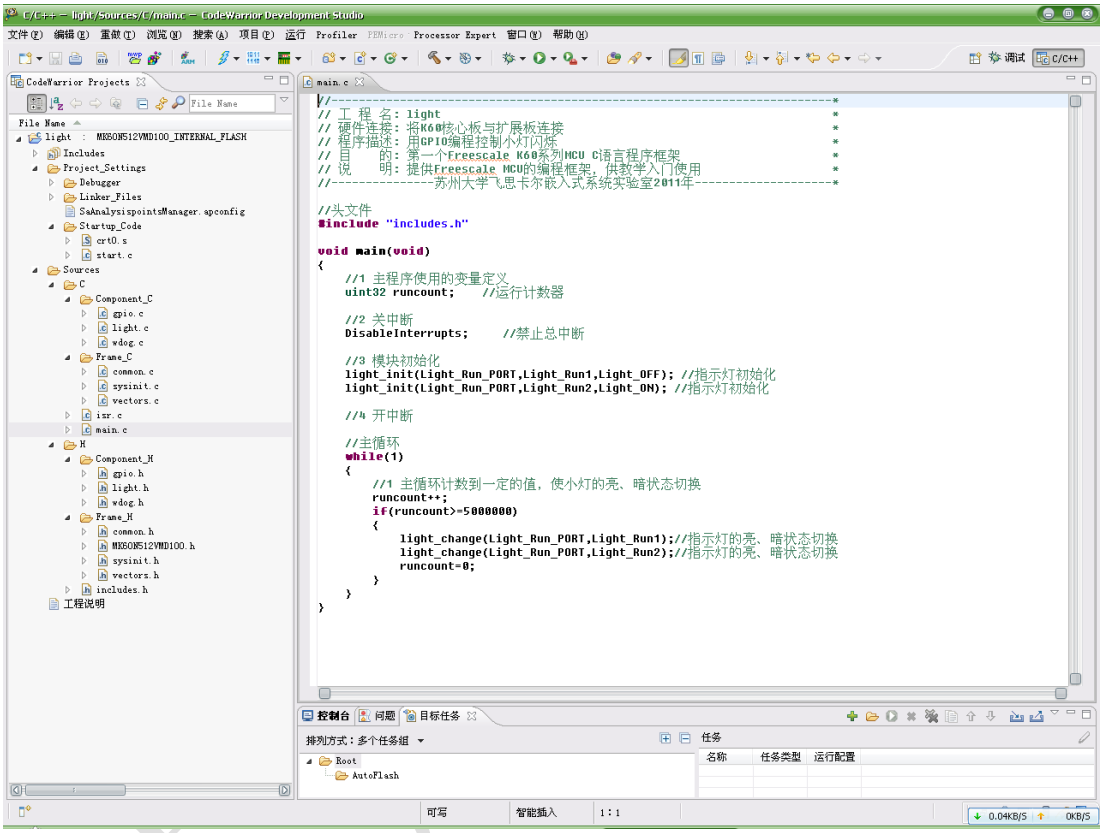

图 3-4 CW10.1 环境运行界面

需要说明的是，安装完毕以后要上网注册以申请使用许可（license key）。无论是下载的软件还是申请到的免费光盘，安装后都要通过因特网注册，以申请使用许可（licenseKey）。这里可通过登录其网站，单击“Request a Key”实现。由于这一注册过程是在网上自动实现的，故只要网络通畅，这个往返过程在数分种之内即可过完成。申请后会通过 E-MAIL 得到一个 License.dat 文件。将该文件复制到相应目录下即可，例如：“C:\Program Files\Freescale\CW MCU v10.1\”。对于免费的特别版本，安装好后用 License.dat 覆盖安装目录下的 License.dat。CW 环境的运行界面如图 3-4 所示。

3.3.3 JTAG 写入器

开发人员可以通过 JTAG 写入器对目标板中的 Flash 进行擦除、写入等操作。将机器码下载到 Flash 后，可以进行程序的运行、调试。图 3-5 给出了写入器的实物图。使用该写入器时，一端连接 PC 的 USB 口，一端连接目标板的 BDM 口。详细的使用说明请见光盘。

3.3.4 MK60N512VMD100 硬件核心板

MK60N512VMD100 硬件核心板如图 3-6 所示，该硬件核心板使用 144 引脚的 MAPBGA

封装，为 4 层电路板。通过扩展板为其供电，扩展板上实现了 K 系列芯片的大部分模块，其中包括 LED，CAN，UART，SD 卡，USB OTG，以太网和 LCD 等模块。扩展板使用两层电路板，提供 12V 电源输入插头，板上使用 LM2576 芯片将 12V 电压转换至 3.3V 电压。LM2576 产生的 3.3V 电源提供 MK60N512VMD100 核心板和扩展板上的所有元件，LM2576 可以提供 3A 的输出电流，足够核心板和外设使用。扩展板实物图见图 3-7 所示。

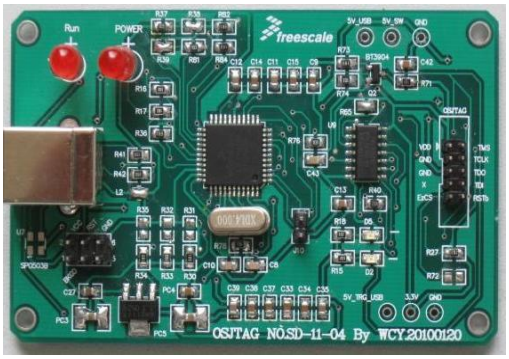

图 3-5 写入器的实物图

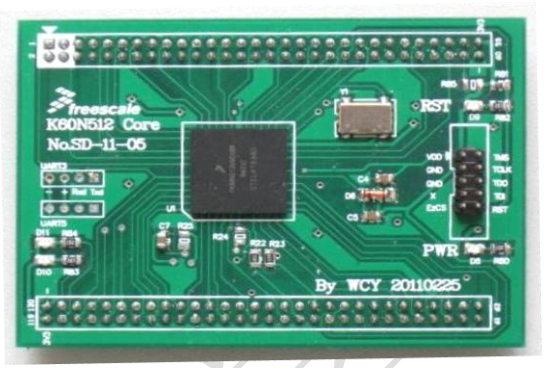

图 3-6 MK60N512VMD100 硬件核心板

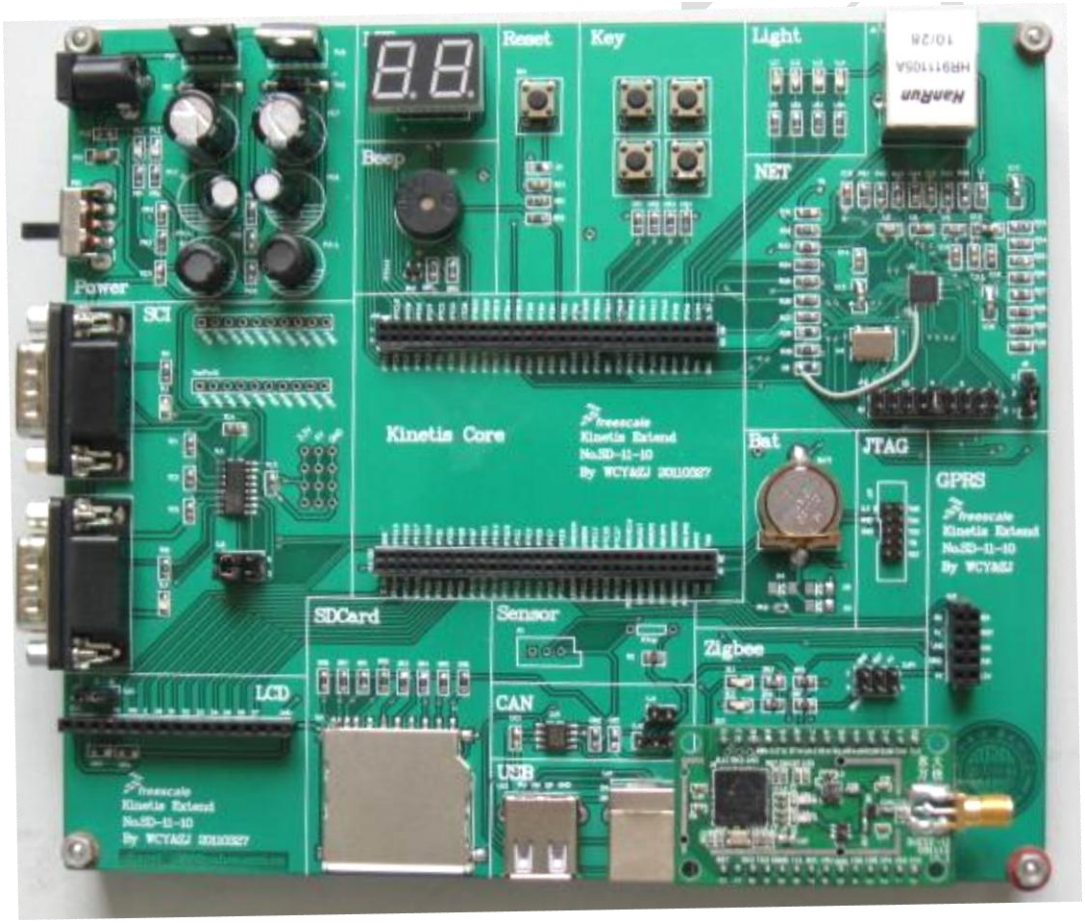

图 3-7 K 系列扩展板

3.4 IAR工程文件组织

嵌入式系统工程往往包含很多文件，如：程序文件、头文件、与编译调试相关的信息文

件、工程说明文件以及工程目标代码文件等。工程文件的合理组织对一个嵌入式系统工程犹为重要，它不但会提高项目的开发效率，同时也降低项目的维护难度。

嵌入式系统工程的文件组织方法以硬件对象为核心来展开，系统中每个对象应包含相关的头文件、程序文件及说明文件等。以硬件对象的方式来组织文件，会使得工程结构清晰，调试定位方便，后期维护容易，这也是嵌入式系统软件工程的基本思想。

3.4.1 工程文件的组织

图 3-8 给出了用 I/O 口控制小灯闪烁工程的树形结构模板，该模板是苏州大学飞思卡尔嵌入式系统实验室专门为 MK60N512VMD100 开发板设计的工程模板。读者在新建工程时可以选择该模板。该模板方便易懂，与 IAR 提供的 DEMO 的工程模板相比文件少，去掉了一些初学者不易理解且不是必须的文件，同时应用底层软件构件的概念改进了程序结构，目的是一开始就引导读者进行规范的文件组织与编程。

新建工程有两种方法，一种是使用工程模板，另一种是使用已存在的工程来建立另外一个工程。

第一种方法的操作步骤如下：

选择 file->WorkSpace，建立工作区，然后选择 Project->Create New Project，弹出 Create New Project 对话框，选择“ARM”，选择编程语言，选中 main，然后点确定，输入文件名，点保存，这样就创建了一个工程。创建工程后需要按照具体的要求来进行配置，这点详见“创建新工程的步骤.docx”。

第二种方法是使用已存的工程来建立另一个工程。当在已有工程的基础上，做另一个项目时，比如在 light 工程的基础上编写 LCD 程序，需要进行如下设置：

（1）更改工程文件夹名为 LCD

（2）更改 light.dep 为 LCD. dep

（3）更改 light.ewd 为 LCD. ewd

（4）更改 light.ewp 为 LCD. Ewp

（5）更改 light.eww 为 LCD. eww

（6）用记事本方式打开 LCD. Eww，代码如下

```
<?xml version="1.0" encoding="iso-8859-1"?>

<workspace>
  <project>
    <path>$WS_DIR$\ light..ewp</path>
  </project>
  <batchBuild/>
</workspace>
```

将其中的 light.ewp 改为 LCD..ewp

（7）打开该工程，你会看到机器码文件还是 light.map 与 light.out，这时进行编译，编译结束后，你会看到 light.map 与 light.out 变为 LCD.map 与 LCD.out。

新建工程时，我们建议采用第二种方式，这种方式比较简单，无需配置，不容易出错。

下面以控制小灯闪烁工程为例，介绍基于 IAR 环境的嵌入式工程文件组织方法。图 3-7 给出了该工程相关源文件的树型结构，可分为“工程配置文件”、 “源程序文件”、“机器码文件”三个部分。

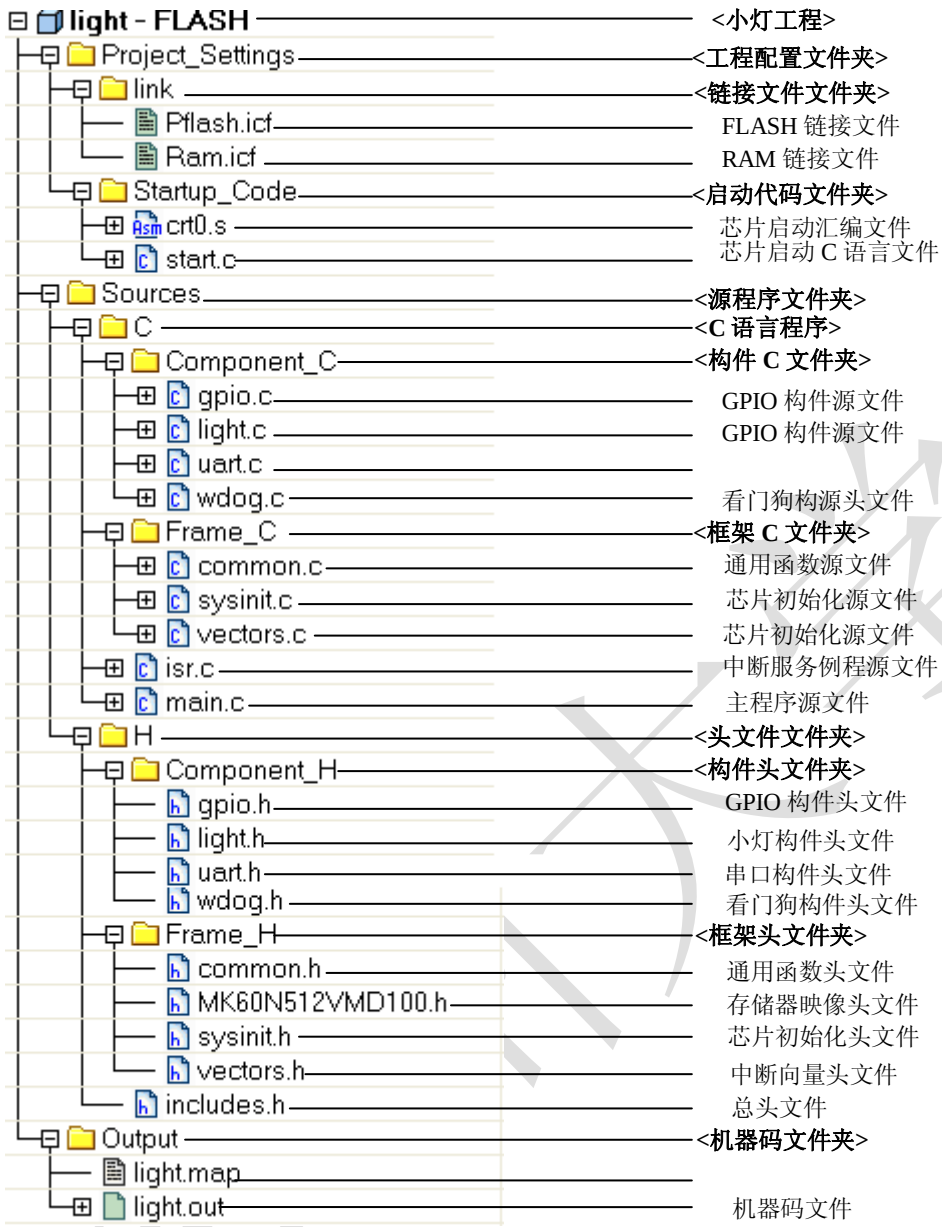

图 3-8 工程相关源文件的树型结构

“工程配置文件”中包含的文件与芯片及工程初始化相关，包括链接文件与启动代码文件。启动代码文件有“crt0.s”与“start.c”，而且前面都有一个⊞，展开后会看到好多文件，这些都是与其相关联的文件。“链接文件”定义了芯片存储器的分配和可执行代码地址空间的分配，包括 Pflash.icf 与 Ram.icf 两个文件，通过修改它们可以将可执行代码链接到芯片 RAM 中或 Flash 中。

“源程序文件夹“包括 C 语言程序、头文件。其中 C 语言程序包括构件 C 文件夹”Component_C“、框架 C 文件夹”Frame_C“、中断服务例程源文件 isr 与主程序源文件 main，头文件包括构件头文件”Component_H“、框架头文件夹”Frame_H“与总头文件 includes.h。系统启动并初始化后，程序根据 main.c 中定义的主循环顺序执行，当遇到中断请求时，转而执行 isr.c 中定义的相应中断处理程序；中断处理结束，则返回中断处继续顺序执行。由于 main.c 和 isr.c 文件反映了软件系统的整体执行流程，故而在工程文件组织时，将它与其余 C 语言程序文件分开管理。”Component_C“与”Component_H“包含构件代码，每个构件都对应一个.c 文件与.h 文件，例如 GPIO.c 与 GPIO.h 文件。以后的章节还会出现“串行

通信”、“键盘”、“LED”、“液晶”等构件。与总体框架程序相关的头文件和源文件分别放在了 Frame_H 和 Frame_C 文件夹中，以归类管理。Frame_H 里包含了 common.h、MK60N512VMD100.h、sysinit.h 与 vectors.h 四个头文件。MK60N512VMD100.h 是芯片寄存器及相关位定义头文件，它被视为芯片的接口文件，没有这个文件，就不可能对该芯片进行任何操作。sysinit.h 与 Frame_C 文件夹中的 sysinit.c 对应，它定义了系统初始化时的基本参数，如系统时钟等，而 sysinit.c 文件则包含实际初始化代码。common.h 与 common.c 对应，它提供常用且基本的软件功能性子函数。

“机器码文件”包括.out 文件与.map 文件，写到 Flash 中的文件为.out 文件，该文件在 IAR 下打不开，不过我们可以打开.srec 文件，它相当于 CodeWarrior 下的.s19 文件。

3.4.2 初始化相关文件

1.启动文件 crt0.s 与 start.c

由于芯片启动代码直接面对内核和硬件控制器进行编程，一般都是用汇编语言实现。在芯片上电复位后，初始化 CPU 各寄存器，关闭中断等，需要用 ARM 的汇编语言编写启动代码（crt0.s 文件），然后跳转到用户 C 程序（start.c），在这里复制中断向量与代码到 RAM 中，初始化芯片时钟，打开中断，然后跳转到 main 函数继续执行。在 ARM 设计开发中，启动代码的编写是一个极重要的过程。启动代码随具体的目标系统和开发系统有所区别，MK60N512VMD100 芯片的启动流程见图 3-9 所示。

1）芯片上电

MK60N512VMD100 允许将中断向量放置在 Flash 或者 RAM 中，但是上电时刻中断向量只能在地址 0x0000_0000 处。上电后，K60 首先从地址 0x0000_0000 处取栈地址，从地址 0x0000_0004 处取复位向量地址，然后跳转至复位向量地址处执行，代码在 vectors.h 处，如下：

```
#define VECTOR_000          (pointer*)__BOOT_STACK_ADDRESS      //     初始化 SP
#define VECTOR_001          __startup   // 0x0000_0004 1 -                初始化 PC
```

2）crt0.s 文件

芯片上电后，转至 crt0.s 文件执行代码如下：

```
_startup
 MOV       r0,#0                          ; 初始化寄存器
MOV        r1,#0
MOV        r2,#0
MOV        r3,#0
MOV        r4,#0
MOV        r5,#0
MOV        r6,#0
MOV        r7,#0
MOV        r8,#0
MOV        r9,#0
MOV        r10,#0
MOV        r11,#0
MOV        r12,#0
CPSIE     i                               ; 屏蔽中断
          import start
          BL          start                        ; 调用 start
```

复位向量中首先清零所有 CPU 通用寄存器，关闭总中断，然后调用 start，依次关闭看门狗、复制中断向量表到 RAM 中、初始化芯片时钟、转到 main 函数继续执行。

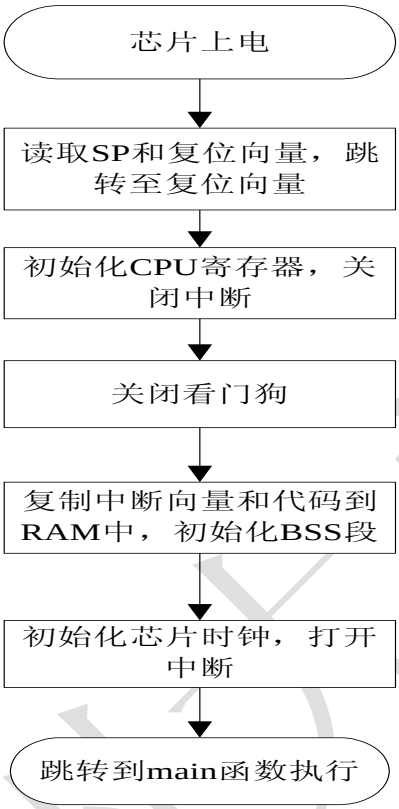

图 3-9 MK60N512VMD100 芯片启动流程

3）start.c 文件

该文件的部分代码如下

```
void start(void)
{
    //关闭看门狗
    wdog_disable();
    //复制中断向量表到 RAM 中
    common_startup();
    //系统设置
    sysinit();
    //进入主函数
    main();
}
```

下面详细讲述下这部分代码。

（1）关闭看门狗

看门狗在嵌入式设计中特别重要，它可以在芯片代码跑飞或死机的情况下复位芯片。嵌入式产品往往 24 小时全天候运行，为了保证程序一直运行正常必须在产品正式发布时使能看门狗。而在程序调试阶段，为了保证程序执行流程，往往先关闭看门狗中断。MK60N512VMD100 的看门狗控制寄存器是只写一次寄存器，即上电后只能对其进行一次写

入，如果想进行多次写入必须首先解锁看门狗。看门狗解锁是向看门狗的解锁寄存器连续写入 0xC520 和 0xD928，两次写入不能超过 20 个时钟周期，否则解锁失败并产生看门狗中断。看门狗解锁后，通过配置看门狗控制寄存器的 WDOGEN 位来关闭看门狗。

如果程序在完成启动之后要使用看门狗，首先要解锁看门狗然后设置 WDOGEN 位来使能看门狗。看门狗使能以后，会在溢出时间超时后产生看门狗复位。程序中必须在溢出超时前“喂狗”。“喂狗”后看门狗模块重新计时，MK60N512VMD100 的“喂狗”是向看门狗刷新寄存器连续写入 0xB480 和 0xA602。两次写入间隔不能超过 20 个时钟周期。源代码请见样例程序中的 wdog.c 文件。

（2）复制中断向量表到 RAM 中

代码在 RAM 中执行的效率比在 Flash 中执行高。通常 ARM 芯片都会将中断向量表复制到 RAM 中。未初始化的数据段（BSS）应该清零。其中涉及到几个 Link 文件中定义的变量，包括：Flash 中断向量表地址 VECTOR_ROM，RAM 中断向量表地址 VECTOR_RAM。这些变量定义在 Link 文件两个.icf 中。源代码请见样例程序中的 startup.c 文件、Pflash.icf 文件与 Ram.icf 文件。

（3）系统设置

芯片主时钟是利用 MCG 模块中的 PLL 模块，通过倍频板上 50MHZ 的有源晶振得到的。在 Kinetis 芯片内部存在 3 种不同时钟：内核时钟，总线时钟和 Flash 时钟。内核时钟是基本时钟，其他时钟均从内核时钟分频得到。Kinetis 手册推荐了 3 种时钟配置方式，分别将内核时钟配置为 50MHZ，96MHZ 和 100MHZ。源代码请见样例程序中的 sysinit.c 文件。

d）跳转至 main 函数执行

上述三步执行结束后，芯片已经进入了正常运行状态，可以执行 main 函数中用户定义的代码了。代码中直接调用 main 函数即可。

到此为止，芯片开始执行用户功能代码，芯片启动阶段结束。

2.芯片相关文件

1）映像寄存器定义文件 MK60N512VMD100.h

MK60N512VMD100.h 中定义了编程时需要访问的外设寄存器，该文件不修改。

2）系统初始化文件 sysinit.c 与 sysinit.h

系统初始化操作是由 sysinit.c 与 sysinit.h 来实现的，具体内容参见样例程序。

3.主程序、中断程序及其它文件

1）总头文件 includes.h 与主程序文件 main.c

includes.h 文件包含主函数（main）文件用到的头文件、外部函数或变量引用、有关常量和全局变量定义以及内部函数声明。而 main.c 文件是工程任务的核心文件，里面包含了一个主循环，对具体事务过程的操作几乎都是添加在该主循环中。

2）中断文件 isr.h 与 isr.c

isr.h 文件包含 isr.c 文件用到的头文件，外部函数以及内部函数声明等，isr.c 文件执行具体的操作，具体内容可参见带有中断的样例程序。

4.链接文件 Pflash.icf 文件与 Ram.icf 文件

链接文件定义了芯片存储器的分配和可执行代码地址空间的分配。可以选择将可执行代码链接到芯片 RAM 中或 Flash 中。具体内容可参见样例程序。

5.机器码文件

在编译链接过程中，IAR 会产生机器码文件.out 文件，这是写入到 Flash 中的文件，在 IAR 环境中打不开。如果在项目配置时选择在编译链接过程产生.srec 文件，那么在控制小灯闪烁的工程中我们可以得到 GPIO.srec 文件，这个文件可以打开，内容以 S 记录格式表示。S 记录格式是 Freescale 公司的十六进制目标代码文件，它将目标程序和数据以 ASCII 码格

式表示，可直接显示和打印。目标文件由若干行 S 记录构成，每行 S 记录可以用 CR/LF/NUL 结尾。一行 S 记录由下列五部分组成，分别说明如下。

（1）类型

表示 S 记录的类型。有 8 种记录类型 S0、S1、S2、S3、S5、S7、S8、S9。这是为了满足不同的编码、解码及传送方式的需求，表 3-1 给出了 S 记录的格式。

表 3-1 S 记录格式

类型	记录长度	地址	编码/数据	校验和
2 字节	2 字节	2、3 或 4 字节	0～n 字节	1 字节

S0—该记录包含 S19 文件的文件名信息。

S1—该记录包含代码/数据以及两个字节存储其代码/数据的存储器首地址。

S2—该记录包含要写到 Flash 的扩展地址处的代码/数据以及三个字节存储其代码/数据的存储器首地址。

S3—该记录包含要写到 Flash 的扩展地址处的代码/数据以及四个字节存储其代码/数据的存储器首地址。

S7—S3 记录的结束记录。

S8—S2 记录的结束记录。

S9—S1 记录的结束记录。

每个 S 记录块都使用唯一的终止记录。

（2）记录长度

表示该记录行中字符对的数目，不包括类型和记录长度。

（3）地址

它可以是 2 个字节、3 个字节或 4 个字节，取决于记录类型。S1 记录、S9 记录均是 2 个字节，S2 记录、S8 记录是 3 个字节，S3 记录、S7 记录是 4 个字节。它表示其后的代码/数据部分将要装入的存储器起始地址。

（4）代码/数据

就是实际的目标代码或数据，这一部分将被下载到目标芯片的存储器并运行。其字节数是由“记录长度”域的实际数值减去地址长度和校验码长度的值而得到的。

（5）校验和

为 1 个字节，它是“记录长度”、“地址”、“代码/数据”三个部分所有字节之和的反码的低 8 位，用于校验。

下面是 01_GPIO 工程中的 GPIO.srec 的部分内容。

S00C00006770696F2E7372656369…

S3151FFF0000F8FF00201104FF1FD127FF1FD127FF1F56

…

S7051FFF291F94

第一行为 S0 记录，表示文件名信息。S0 之后的 0C 是十六进制数（十进制数 12），表示后面有 12 个字节的数据；随后的"0000"是 2 字节地址，"0000"表示本行信息不是程序/数据，不需要装入存储空间；最后的 69 是本记录的校验和。其中 S3151FFF0000F8FF00201104FF1FD127FF1FD127FF1F56 的前两个符号 S3 表示这一行是 S3 记录，其后的“15”是十六进制数（十进制数的 21），表示在此行其后有 21 个字节的数据，包括 4 个字节的地址 1FFF0000、16 个字节的代码/数据，最后字节 56 为校验和。该行记录所表示的实际代码/数据 F8FF00201104FF1FD127FF1FD127FF1F 将被装入起始地址为 1FFF0000 的 MCU 存储器中。

最后一行是 S7 记录，S7 之后的 05 是十六进制 0x05，表示其后有 5 个字节的/数据。1FFF291F 为 4 个字节的地址，94 是校验和。

3.5 CW工程文件组织

CW 工程文件组织与 IAR 工程文件组织基本一样，但略有不同，下面简单阐述一下。

3.5.1 工程文件的组织

图 3-10 给出了用 I/O 口控制小灯闪烁工程的树形结构模板，该模板是苏州大学飞思卡尔嵌入式系统实验室专门为 MK60N512VMD100 开发板设计的工程模板。读者在编程时可以选择该模板。该模板方便易懂，与 CW 提供的 DEMO 工程模板相比，去掉了一些初学者不易理解且不是必须的文件，同时应用底层软件构件的概念改进了程序结构，目的是一开始就引导读者进行规范的文件组织与编程。

新建工程有两种方法，一种是使用工程模板，另一种是使用已存在的工程来建立另外一个工程。

第一种方法的操作步骤如下：

选择 file->new->Bareboard Project，弹出 New Bareboard Project 对话框，然后根据芯片型号来选择配置就创建了一个工程，详见“创建新工程的步骤.docx”。

第二种方法是使用已存的工程来建立另一个工程。当在已有工程的基础上，做另一个项目时，只需更改工程名即可。

新建工程时，我们建议采用第二种方式，这种方式比较简单，无需配置，不容易出错。

下面以控制小灯闪烁工程为例，介绍基于 CW 环境的嵌入式工程文件组织方法。图 3-9 给出了该工程相关源文件的树型结构，可分为“头文件路径”、“工程配置文件”、“输出文件”、“应用程序文件”4 大部分。

“头文件路径”为工程配置后自动生成的文件，不用改动。

“输出文件”，包含.afx 和 S19 等格式的目标机器码。

“工程配置文件”包含与调试相关的配置文件，链接文件，启动代码文件。

“应用程序文件”包含通用函数，构件文件，主程序文件，中断服务例程文件等。

3.5.2 初始化相关文件

这部分与 IAR 环境基本一致，只是机器码文件稍有不同而已。在编译链接过程中，CW 会产生机器码文件.afx 文件，这是写入到 Flash 中的文件，在 CW 环境中打不开。如果在项目配置时选择在编译链接过程产生.S19 文件，那么在控制小灯闪烁的工程中我们可以得到 Light.afx.S19 文件，这个文件可以打开，内容以 S 记录格式表示。

图 3-10 工程相关源文件的树型结构

3.6 第一个应用实例：控制小灯闪烁

本书用 MK60N512VMD100 控制发光二极管指示灯的例子开始我们的程序之旅，程序中使用了 GPIO 构件来编写指示灯程序。指示灯是最简单不过的硬件对象了，当灯两端引脚上有足够高的正向压降时，它就会发光。在本书的工程实例中，灯的正端引脚接 MK60N512VMD100 的普通 I/O 口，负端引脚过电阻接地。当在 I/O 引脚上输出高或低电平时，指示灯就会亮或暗。

3.6.1 GPIO 构件

GPIO 引脚可以被定义成输入、输出两种情况。若是输入，程序需要获得引脚的状态（逻辑 1 或 0）。若是输出，程序可以设置引脚状态（逻辑 1 或 0）。MCU 的 GPIO 引脚分为许多端口（Port），每个口有若干引脚。为了实现对所有 GPIO 引脚统一编程，设计了 GPIO 构件（由 GPIO.h、GPIO.c 两个文件组成）。这样，要使用 GPIO 构件，只需要将这两个文件加入到所建工程中，方便了对 GPIO 的编程操作。实际上，若只是使用构件，只需看头文件中的相关函数说明。

1．GPIO 构件的头文件 gpio.h

```
//-----------------------------------------------------------------------*
// 文件名: gpio.h                                                        *
// 说  明: gpio驱动头文件                                                *
//-----------------------------------------------------------------------*

#ifndef __GPIO_H__
#define __GPIO_H__

    //1 头文件
    #include "common.h"

    //2 宏定义
    //2.1 端口宏定义
    #define PORTA PTA_BASE_PTR
    #define PORTB PTB_BASE_PTR
    #define PORTC PTC_BASE_PTR
    #define PORTD PTD_BASE_PTR
    #define PORTE PTE_BASE_PTR

    //3 函数声明
    //-----------------------------------------------------------------------*
    //函数名: gpio_init                                                     *
    //功  能: 初始化gpio                                                    *
    //参  数: port:端口名                                                   *
    //        index:指定端口引脚                                            *
    //        dir:引脚方向,0=输入,1=输出                                    *
    //        data:初始状态,0=低电平,1=高电平                               *
    //返  回: 无                                                            *
    //说  明: 无                                                            *
    //-----------------------------------------------------------------------*
    void gpio_init (GPIO_MemMapPtr port, int index, int dir,int data);

    //-----------------------------------------------------------------------*
    //函数名: gpio_ctrl                                                     *
    //功  能: 设置引脚状态                                                  *
    //参  数: port:端口名                                                   *
    //        index:指定端口引脚                                            *
    //        data: 状态,0=低电平,1=高电平                                  *
    //返  回: 无                                                            *
    //说  明: 无                                                            *
    //-----------------------------------------------------------------------*
    void gpio_ctrl (GPIO_MemMapPtr port, int index, int data);
```

```
    //-------------------------------------------------------------------------*
    //函数名: gpio_reverse                                                     *
    //功  能: 改变引脚状态                                                     *
    //参  数: port:端口名;                                                     *
    //        index:指定端口引脚                                               *
    //返  回: 无                                                               *
    //说  明: 无                                                               *
    //-------------------------------------------------------------------------*
    void gpio_reverse (GPIO_MemMapPtr port, int index);

#endif
```

2．GPIO 构件的程序文件 gpio.c

```
//-------------------------------------------------------------------------*
// 文件名: gpio.c                                                          *
// 说  明: gpio驱动程序文件                                                *
//-------------------------------------------------------------------------*

#include "gpio.h"     //包含gpio头文件

//-------------------------------------------------------------------------*
//函数名: gpio_init                                                        *
//功  能: 初始化gpio                                                       *
//参  数: port:端口名                                                      *
//        index:指定端口引脚                                               *
//        dir:引脚方向,0=输入,1=输出                                       *
//        data:初始状态,0=低电平,1=高电平                                  *
//返  回: 无                                                               *
//说  明: 无                                                               *
//-------------------------------------------------------------------------*
void gpio_init (GPIO_MemMapPtr port, int index, int dir,int data)
{
    PORT_MemMapPtr p;
    switch((uint32)port)
    {
    case 0x400FF000u:
        p = PORTA_BASE_PTR;
        break;
    case 0x400FF040u:
        p = PORTB_BASE_PTR;
        break;
    case 0x400FF080u:
        p = PORTC_BASE_PTR;
        break;
    case 0x400FF0C0u:
        p = PORTD_BASE_PTR;
        break;
    case 0x400FF100u:
        p = PORTE_BASE_PTR;
        break;
    default:
        break;
    }
    PORT_PCR_REG(p,index)=(0|PORT_PCR_MUX(1));
```

```
    if(dir == 1)//output
    {
        GPIO_PDDR_REG(port) |= (1<<index);
        if(data == 1)//output
GPIO_PDOR_REG(port) |= (1<<index);
else
GPIO_PDOR_REG(port) &= ~(1<<index);
    }

    else
        GPIO_PDDR_REG(port) &= ~(1<<index); }
```

```
//-------------------------------------------------------------------------*
//函数名: gpio_ctrl                                                        *
//功  能: 设置引脚状态                                                     *
//参  数: port:端口名                                                      *
//        index:指定端口引脚                                               *
//        data: 状态,0=低电平,1=高电平                                     *
//返  回: 无                                                               *
//说  明: 无                                                               *
//-------------------------------------------------------------------------*
void gpio_ctrl (GPIO_MemMapPtr port, int index, int data)
{
    if(data == 1)//output
        GPIO_PDOR_REG(port) |= (1<<index);
    else
        GPIO_PDOR_REG(port) &= ~(1<<index);
}
//-------------------------------------------------------------------------*
//函数名: gpio_reverse                                                     *
//功  能: 改变引脚状态                                                     *
//参  数: port:端口名;                                                     *
//        index:指定端口引脚                                               *
//返  回: 无                                                               *
//说  明: 无                                                               *
//-------------------------------------------------------------------------*
void gpio_reverse (GPIO_MemMapPtr port, int index)
{
    GPIO_PDOR_REG(port) ^= (1<<index);
}
```

3.6.2 Light 构件

控制指示灯的亮或暗，通过调用 GPIO 构件完成。设有两盏灯，分别为运行指示灯 1、运行指示灯 2，分别叫做 Light_Run1、Light_Run2。它们所接在的 MCU 的 GPIO 口的名字叫做 Light_Run_PORT。它们具体接在 MCU 的哪个端口，哪个引脚，只要在 light.h 中给出具体宏定义就可以了。

1.Light 构件的头文件 light.h

```
//-------------------------------------------------------------------------*
// 文件名: light.h                                                         *
// 说  明: 指示灯驱动程序头文件                                            *
//-------------------------------------------------------------------------*
```

```
#ifndef LIGHT_H_
#define LIGHT_H_

    //1 头文件
    #include "common.h"
    #include "gpio.h"

    //2 宏定义
    //2.1 灯控制引脚定义
    #define Light_Run_PORT PORTC     //运行指示灯使用的端口
    #define Light_Run1      13       //运行指示灯使用的引脚
    #define Light_Run2      14       //运行指示灯使用的引脚

    //2.2 灯状态宏定义
    #define Light_ON        0        //灯亮(对应低电平)
    #define Light_OFF       1        //灯暗(对应高电平)


 //3                函                数                声                明
 //-------------------------------------------------------------------------*
//函数名: light_init                                                       *
//功  能: 初始化指示灯状态                                                 *
 //参  数: port:端口名                                                      *
 //        name:指定端口引脚号                                              *
 //        state:初始状态,1=高电平,0=低电平                                 *
 //返  回: 无                                                               *
 //说  明: 调用GPIO_Init函数                                                *
    //-------------------------------------------------------------------------*
       void light_init(GPIO_MemMapPtr port,int name,int state);
//-------------------------------------------------------------------------*
 //函数名: Light_control                                                    *
 //功  能: 控制灯的亮和暗                                                   *
 //参  数: port:端口名                                                      *
 //        name:指定端口引脚号                                              *
 //        state:状态,1=高电平,0=低电平                                     *
 //返  回: 无                                                               *
 //说  明: 调用GPIO_Set函数                                                 *
    //-------------------------------------------------------------------------*
      void light_control(GPIO_MemMapPtr port,int name,int state);
 //-------------------------------------------------------------------------*
 //函数名: Light_change                                                     *
 //功  能: 状态切换:原来"暗",则变"亮";原来"亮",则变"暗"                    *
 //参  数: port:端口名                                                      *
 //        name:指定端口引脚号                                              *
 //返  回: 无                                                               *
 //    说    明    :    调    用    GPIO_Get    、    GPIO_Set    函    数
 //-------------------------------------------------------------------------*
      void light_change(GPIO_MemMapPtr port,int name);
 #endif
```

2.Light 构件的程序文件 light.c

```
//-------------------------------------------------------------------------*
// 文件名: light.c                                                         *
// 说  明: 小灯驱动函数文件                                                *
//-------------------------------------------------------------------------*
```

```
#include "light.h"                  //指示灯驱动程序头文件

//-------------------------------------------------------------------------*
//函数名: light_init                                                       *
//功  能: 初始化指示灯状态                                                 *
//参  数: port:端口名                                                      *
//        name:指定端口引脚号                                              *
//        state:初始状态,1=高电平,0=低电平                                 *
//返  回: 无                                                               *
//说  明: 调用gpio_init函数                                                *
//-------------------------------------------------------------------------*
void light_init(GPIO_MemMapPtr port,int name,int state)
{
gpio_init(port,name,1,state);//初始化指示灯
}
//-------------------------------------------------------------------------*
//函数名: light_control                                                    *
//功  能: 控制灯的亮和暗                                                   *
//参  数: port:端口名                                                      *
//        name:指定端口引脚号                                              *
//        state:状态,1=高电平,0=低电平                                     *
//返  回: 无                                                               *
//说  明: 调用gpio_ctrl函数                                                *
//-------------------------------------------------------------------------*
void light_control(GPIO_MemMapPtr port,int name,int state)
{
gpio_ctrl(port,name,state);    //控制引脚状态
}

//-------------------------------------------------------------------------*
//函数名: light_change                                                     *
//功  能: 状态切换:原来"暗",则变"亮";原来"亮",则变"暗"                     *
//参  数: port:端口名                                                      *
//        name:指定端口引脚号                                              *
//返  回: 无                                                               *
//说  明: 调用gpio_reverse函数                                             *
//-------------------------------------------------------------------------*
void light_change(GPIO_MemMapPtr port,int name)
{
gpio_reverse(port,name);
}
```

3.6.3 Light 测试工程主程序

在 includes.h 文件中需要包含 GPIO.h，这样在该工程中就可以调用 GPIO 构件的接口函数。首先调用 gpio_init 函数，初始化所需的每一盏指示灯。随后，通过 gpio_reverse 函数将引脚电平取反，就能够在程序运行时，较明显的看到指示灯闪烁的现象。代码如下：

```
//-------------------------------------------------------------------------*
// 工 程 名: light                                                         *
// 硬件连接: 将K60核心板与扩展板连接                                       *
// 程序描述: 用GPIO编程控制小灯闪烁                                        *
// 目    的: 第一个Freescale K60系列MCU C语言程序框架                     *
// 说    明: 提供Freescale MCU的编程框架，供教学入门使用                   *
//---------------苏州大学飞思卡尔嵌入式系统实验室2011年--------------------*
```

```
//头文件
#include "includes.h"

//全局变量声明

//主函数
void main(void)
{
    //1 主程序使用的变量定义
    uint32 runcount;                                    //运行计数器

    //2 关中断
    DisableInterrupts;     //禁止总中断

    //3 模块初始化
    light_init(Light_Run_PORT,Light_Run1,Light_OFF); //指示灯初始化
    light_init(Light_Run_PORT,Light_Run2,Light_ON); //指示灯初始化

    //4 开中断

    //主循环
    while(1)
    {
        //1 主循环计数到一定的值，使小灯的亮、暗状态切换
        runcount++;
        if(runcount>=5000000)
        {
            light_change(Light_Run_PORT,Light_Run1);//指示灯的亮、暗状态切换
            light_change(Light_Run_PORT,Light_Run2);//指示灯的亮、暗状态切换
            runcount=0;
        }
    }
}
```

3.7 理解第一个C工程的执行过程

当 MK60N512VMD100 芯片上电复位后或热复位后，系统程序的执行流程如下：从复位向量处取出程序执行的首地址，跳转并按地址执行；执行 crt0.s 文件中的__startup，复位所有的通用寄存器，关闭总中断，然后调用 start.c 文件，进行系统初始化，最终跳转到 main 主函数入口继续执行；这部分在“3.4.2 初始化相关文件”的第一部分“1.启动文件 crt0.s 与 start.c”已经讲述，这里不再赘述。

第4章 异步串行通信

目前几乎所有的台式电脑都带有 9 芯的异步串行通信口，简称串行口或 COM 口。由于历史的原因，通常所说的串行通信就是指异步串行通信。USB、以太网等也用串行方式通信，但与这里所说的异步串行通信物理机制不同。

有的台式电脑带有两个串行口：COM1、COM2 口，部分笔记本电脑也带有串行口。随着 USB 接口的普及，串行口的地位逐渐降低，但是作为设备间简便的通信方式，在相当长的时间内，串行口还不会消失，在市场上也可很容易的购买到 USB 到串行口的转接器。因为简单且常用的串行通信只需要三根线（发送线、接收线和地线），所以串行通信仍然是 MCU 与外界通信的简便方式之一。

实现异步串行通信功能的模块在一部分 MCU 中被称为通用异步收发器（Universal Asynchronous Receiver/Transmitters，UART），在另一些 MCU 中被称为串行通信接口（Serial Communication Interface，SCI）。串行通信接口可以将终端或个人计算机连接到 MCU，也可将几个分散的 MCU 连接成通信网络。

本章主要介绍 MK60N512VMD100 的 UART 模块的工作原理以及编程实例，这些编程实例都使用了基于构件的编程思想，读者在阅读时可以仔细体会，以求得对编程方法更深刻的理解。下文所出现的 UART 字眼，在没有其他说明的情况下，都是特指 MK60N512VMD100 的 UART 模块。

4.1 异步串行通信的基础知识

本节简要概括了串行通信中常用的基本概念，为学习 MCU 的串行接口编程做准备。对于已经了解这方面知识的读者，可以略读本节。

4.1.1 基本概念

“位”（bit）是单个二进制数字的简称，是可以拥有两种状态的最小二进制值，分别用“0”和“1”表示。在计算机中，通常一个信息单位用 8 位二进制表示，称为一个“字节”（Byte）。串行通信的特点是：数据以字节为单位，按位的顺序（例如最高位优先）从一条传输线上发送出去。这里至少涉及到以下几个问题：第一，每个字节之间是如何区分开的？第二，发送一位的持续时间是多少？第三，怎样知道传输是正确的？第四，可以传输多远？这些问题属于串行通信的基本概念。串行通信分为异步通信与同步通信两种方式，本节主要给出异步串行通信的一些常用概念。正确理解这些概念，对串行通信编程是有益的。

1．异步串行通信的格式

在 MCU 的英文芯片手册上，通常说的异步串行通信采用的是 NRZ 数据格式，英文全称是：“standard non-return-zero mark/space data format”，可以译为：“标准不归零传号/空号数据格式”。这是一个通信术语，“不归零”的最初含义是：用负电平表示一种二进制值，正电平表示另一种二进制值，不使用零电平。“mark/space”即“传号/空号”分别是表示两种状态的物理名称，逻辑名称记为“1/0”。对学习嵌入式应用的读者而言，只要理解这种格式只有“1”、“0”两种逻辑值就可以了。图 4-1 给出了 8 位数据、无校验情况的传送格式。

图 4-1 串行通信数据格式

这种格式的空闲状态为“1”，发送器通过发送一个“0”表示一个字节传输的开始，随后是数据位（在 MCU 中一般是 8 位或 9 位，可以包含校验位）。最后，发送器发送 1 到 2 位的停止位，表示一个字节传送结束。若继续发送下一字节，则重新发送开始位，开始一个新的字节传送。若不发送新的字节，则维持“1”的状态，使发送数据线处于空闲。从开始位到停止位结束的时间间隔称为一帧（frame)。所以，也称这种格式为帧格式。

每发送一个字节，都要发送“开始位”与“停止位”，这是影响异步串行通信传送速度的因素之一。同时因为每发送一个字节，必须首先发送“开始位”，所以称之为“异步”（Asynchronous）通信。

2．串行通信的波特率

位长（Bit Length），也称为位的持续时间（Bit Duration）。其倒数就是单位时间内传送的位数。人们把每秒内传送的位数叫做波特率（Baud Rate）。波特率的单位是：位/秒，记为 bps。bps 是英文 bit per second 的缩写，习惯上这个缩写不用大写，而用小写。通常情况下，波特率的单位可以省略。

通常使用的波特率有 600、900、1200、1800、2400、4800、9600、19200、38400、57600、115200 等。在包含开始位与停止位的情况下，发送一个字节需 10 位，很容易计算出，在各波特率下，发送 1K 字节所需的时间。显然，这个速度相对于目前许多通信方式而言是慢的，那么，异步串行通信的速度能否提得很高呢？答案是不能。因为随着波特率的提高，位长变小，以致于很容易受到电磁源的干扰，通信就不可靠了。当然，还有通信距离问题，距离小，可以适当提高波特率，但这样毕竟提高的幅度非常有限，达不到大幅度提高的目的。

3．奇偶校验

在异步串行通信中，如何知道传输是正确的？最常见的方法是增加一个位（奇偶校验位），供错误检测使用。字符奇偶校验检查（Character Parity Checking）称为垂直冗余检查（Vertical Redundancy Checking，VRC），它是为每个字符增加一个额外位使字符中“1”的个数为奇数或偶数。奇数或偶数依据使用的是“奇校验检查”还是“偶校验检查”而定。当使用“奇校验检查”时，如果字符数据位中“1”的数目是偶数，校验位应为“1”，如果“1”的数目是奇数，校验位应为“0”。当使用“偶校验检查”时，如果字符数据位中“1”的数目是偶数，则校验位应为“0”，如果是奇数则为“1”。

这里列举奇偶校验检查的一个实例，看看 ASCII 字符“R”，其位构成是 1010010。由于字符“R”中有 3 个位为“1”，若使用奇校验检查，则校验位为 0；如果使用偶校验检查，则校验位为 1。

在传输过程中，若有 1 位（或奇数个数据位）发生错误，使用奇偶校验检查，可以知道发生传输错误。若有 2 位（或偶数个数据位）发生错误，使用奇偶校验检查，就不能知道已经发生了传输错误。但是奇偶校验检查方法简单，使用方便，发生 1 位错误的概率远大于 2 位的概率，所以“奇偶校验”这种方法还是最为常用的校验方法。几乎所有 MCU 的串行异步通信接口，都提供这种功能。

4．串行通信的传输方式

在串行通信中，经常用到“单工”、“双工”、“半双工”等术语。它们是串行通信的不同传输方式。下面简要介绍这些术语的基本含义。

（1）单工（Simplex）：数据传送是单向的，一端为发送端，另一端为接收端。这种传输方式中，除了地线之外，只要一根数据线就可以了。有线广播就是单工的。

（2）全双工（Full-duplex）：数据传送是双向的，且可以同时接收与发送数据。这种传输方式中，除了地线之外，需要两根数据线，站在任何一端的角度看，一根为发送线，另一根为接收线。一般情况下，MCU 的异步串行通信接口均是全双工的。

（3）半双工（Half-duplex）：数据传送也是双向的，但是在这种传输方式中，除地线之外，一般只有一根数据线。任何时刻，只能由一方发送数据，另一方接收数据，不能同时收发。

4.1.2 RS-232C 总线标准

现在回答“可以传输多远”这个问题。MCU 引脚输入/输出一般使用 TTL（Transistor Transistor Logic）电平，即晶体管-晶体管逻辑电平。而 TTL 电平的“1”和“0”的特征电压分别为 2.4V 和 0.4V（目前使用 3V 供电的 MCU 中，该特征值有所变动），即大于 2.4V 则识别为“1”，小于 0.4V 则识别为“0”。它适用于板内数据传输。若用 TTL 电平将数据传输到 5m 之外，那么可靠性就很值得考究了。为使信号传输得更远，美国电子工业协会 EIA（Electronic Industry Association）制订了串行物理接口标准 RS-232C。RS-232C 采用负逻辑，-15V～-3V 为逻辑“1”，+3V～+15V 为逻辑“0”。RS-232C 最大的传输距离是 30m，通信速率一般低于 20Kbps。当然，在实际应用中，也有人用降低通信速率的方法，通过 RS-232 电平，将数据传送到 300m 之外，这是很少见的，且稳定性很不好。

RS-232C 总线标准最初是为远程数据通信制订的，但目前主要用于几米到几十米范围内的近距离通信。有专门的书籍介绍这个标准，但对于一般的读者，不需要掌握 RS-232C 标准的全部内容，只要了解本节介绍的这些基本知识就可以使用 RS-232。目前一般的 PC 机均带有 1 到 2 个串行通信接口，人们也称之为 RS-232 接口，简称“串口”，它主要用于连接具有同样接口的室内设备。早期的标准串行通信接口是 25 芯插头，这是 RS-232C 规定的标准连接器（其中：2 条地线，4 条数据线，11 条控制线，3 条定时信号，其余 5 条线备用或未定义）。

后来，人们发现在计算机的串行通信中，25 芯线中的大部分并不使用，逐渐改为使用 9 芯串行接口。一段时间内，市场上还有 25 芯与 9 芯的转接头，方便了两种不同类型之间的转换。后来，使用 25 芯串行插头极少见到，25 芯与 9 芯转接头也极少有售。因此，目前几乎所有计算机上的串行口都是 9 芯接口。图 4-2 给出了 9 芯串行接口的排列位置，相应引脚含义见表 4-1。

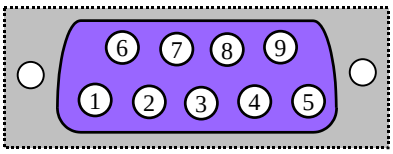

图 4-2 9 芯串行接口排列

在 RS-232 通信中，常常使用精简的 RS-232 通信，通信时仅使用 3 根线：RXD（接收线）、TXD（发送线）和 GND（地线）。其他为进行远程传输时接调制解调器之用，有的也可作为硬件握手信号，初学时可以忽略这些信号的含义。

表 4-1 9 芯串行接口引脚含义表

引脚号	功　能	引脚号	功　能
1	接收线信号检测(载波检测DCD)	6	数据通信设备准备就绪(DSR)

2	接收数据线(RXD)	7	请求发送(RTS)
3	发送数据线(TXD)	8	允许发送(CTS)
4	数据终端准备就绪(DTR)	9	振铃指示
5	信号地(SG)		

4.1.3 电平转换电路原理

在 MCU 中，若用 RS-232C 总线进行串行通信，则需外接电路实现电平转换。在发送端，需要用驱动电路将 TTL 电平转换成 RS-232C 电平；在接收端，需要用接收电路将 RS-232C 电平转换为 TTL 电平。电平转换器不仅可以由晶体管分立元件构成，也可以直接使用集成电路。目前使用 MAX232 芯片较多，该芯片使用单一+5V 电源供电实现电平转换。图 4-3 给出了 MAX232 的引脚说明。

引脚含义简要说明如下：

Vcc（16 脚）：正电源端，一般接+5V

GND（15 脚）：地

VS+（2 脚）：VS+=2Vcc-1.5V=8.5V

VS-（6 脚）：VS-=-2Vcc-1.5V=-11.5V

C2+、C2-（4、5 脚）：一般接 1μF 的电解电容

C1+、C1-（1、3 脚）：一般接 1μF 的电解电容

输入输出引脚分两组，基本含义见表 4-2。在实际使用时，若只需要一路串行通信接口，可以使用其中的任何一组。利用 MAX232 将 TTL 电平转换为 RS-232 电平的电路接法稍后结合串行通信接口的外围硬件电路讨论。

焊接到 PCB 板上的 MAX232 芯片检测方法：正常情况下，（1）T1IN=5V，则 T1OUT=-9V；T1IN=0V，则 T1OUT=9V。（2）将 R1IN 与 T1OUT 相连，令 T1IN=5V，则 R1OUT=5V；令 T1IN=0V，则 R1OUT=0V。

所有型号 MCU 的串行通信接口，都具有发送引脚 TxD、接收引脚 RxD，它们是 TTL 电平引脚。要利用这两个引脚与外界实现异步串行通信，还必须将 TTL 电平转为 RS-232 电平，这可利用上节介绍的 MAX232 来完成。本节从通用角度讨论串行通信接口的电路设计、基本编程结构与编程原理，为实际编程做准备。

表 4-2 MAX232 芯片输入输出引脚分类与基本接法

组别	TTL电平引脚	方向	典型接口	232电平引脚	方向	典型接口
1	11	输入	接MCU的TXD	13	输入	接到9芯接口的2脚RXD
	12	输出	接MCU的RXD	14	输出	接到9芯接口的3脚TXD
2	10	输入	接MCU的TXD	8	输入	接到9芯接口的2脚RXD
	9	输出	接MCU的RXD	7	输出	接到9芯接口的3脚TXD

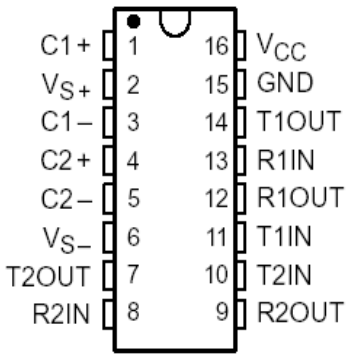

图 4-3 MAX232 引脚

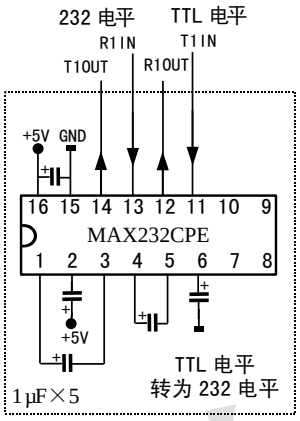

图 4-4 串行通信接口电平转换电路

具有串行通信接口的 MCU，一般具有发送引脚（TXD）与接收引脚（RXD），不同公司或不同系列的 MCU，使用的引脚缩写名可能不一致，但含义相同。串行通信接口的外围硬件电路，主要目的是将 MCU 的发送引脚 TXD 与接收引脚 RXD 的 TTL 电平，通过 RS-232 电平转换芯片转换为 RS-232 电平。图 4-4 给出了基本串行通信接口的电平转换电路。

基本工作过程是：

发送过程：MCU 的 TXD（TTL 电平）经过 MAX232 的 11 脚(T1IN)送到 MAX232 内部，在内部 TTL 电平被“提升”为 232 电平，通过 14 脚(T1OUT)发送出去。

接收过程：外部 232 电平经过 MAX232 的 13 脚（R1IN）进入到 MAX232 的内部，在内部 232 电平被“降低”为 TTL 电平，经过 12 脚（R1OUT）送到 MCU 的 RXD，进入 MCU 内部。

进行 MCU 的串行通信接口编程时，只针对 MCU 的发送与接收引脚，与 MAX232 无关，MAX232 只是起到电平转换作用。

4.2 MK60N512VMD100的UART模块功能描述

MK60N512VMD100 的通用异步收发器 UART，支持全双工的数据传输，可编程 8 位或者 9 位数据格式，采用标准不归零传号/空号（NRZ）格式，可以选择通过配置波特率采用可编程脉冲宽度的 IrDA 1.4 归零逆转（RZI）格式。基于模块时钟频率的 32 分之一，有 13 位波特率的选择。可以独立地启用发送器和接收器，分别设置发送器与接收器的极性，每一个发送和接收可支持 1、4、8、16、32、64 和 128 数据字的缓冲区，发送与接收有独立的 FIFO 结构。支持 SIM 卡和智能卡接口的 ISO 7816 协议。12 个标志符的中断驱动操作。UART 发送器的硬件可产生并发送奇偶校验位，而接收器的奇偶校验硬件则能据此确保接收数据的完整性。具有接收器帧错误检测功能，带有 DMA 接口。

本节主要阐述 MK60N512VMD100 的 UART 模块的结构及功能，包括波特率的计算方法、发送器和接收器等。

MK60N512VMD100 包括 6 个相同且独立的 UART 模块，每个模块都含有相互独立的发送器和接收器。

1．外部引脚

UART 的外部引脚有：

（1）发送数据引脚：UTXDn

（2）接收数据引脚：URXDn

引脚名中的“U”是 UART 的简写，“n”表示模块的编号，取 0～5。通常情况串行通信只使用发送数据引脚 UTXDn 与接收数据引脚 URXDn。

2．波特率发生器

UART0 和 UART1 时钟源为内核时钟，UART2～UART5 的时钟源为外设时钟（总线时钟）。波特率由一个 13 位的模数计数器和一个 5 位的分数微调计数器共同决定。13 位 SBR[SBR]范围 1～8191，它决定了模块的时钟分频。微调计数器给波特率时钟增加一个细微的延时，以便匹配系统波特率。波特率时钟与模块时钟同步并驱动接收器。计算公式如下：

UAER 波特率 = UART 模块时钟/(16*(SBR[SBR]+BRFD))

3．发送器的内部结构

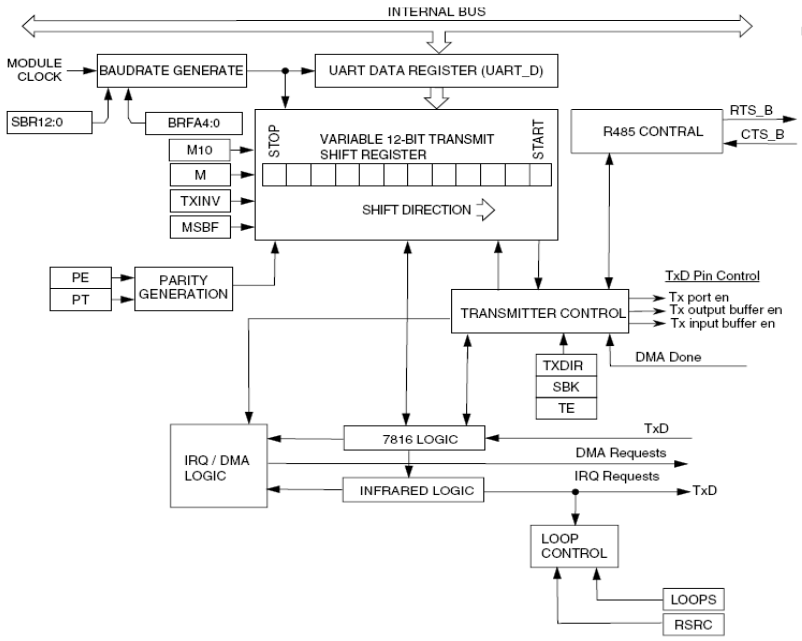

图 4-5 发送器的内部结构

图 4-5 为发送器的内部结构。UART 发送器可以容纳 8、9 或 10 位数据字符。C1[M]和 C1[PE]位和 C4[M10]的状态决定了数据字符的长度。

MCU 通过数据寄存器将数据写入到发送数据缓冲区，然后发送器进行移位发送。当发送结束后，会置位发送缓冲区空标志位(S1[TDRE])，同时也可以根据设置决定是否产生中断。

4．接收器的内部结构

图 4-6 为接收器的内部结构。UART 接收器可以容纳 8、9 或 10 位数据字符。C1[M]和 C1[PE]位和 C4[M10]的状态决定数据字符的长度。

在 UART 接收期间，接收移位寄存器从异步接收器输入信号移进一个帧。一个完整的帧移进接收移位寄存器后，帧的数据部分发送到 UART 接收缓冲区中。另外，接收进程期间可能的噪音和奇偶校验错误标志也被拷贝到了 UART 接收缓冲区中。接收数据缓冲区通

过数据寄存器和 C3[T8]寄存器访问。如果接收缓冲区中的数据字数目等于或多于 RWFIFO[RXWATER]指定的数目，那么 S1[RDRF]标志位会被置 1。如果 C2[RIE]也设为 1，则会产生一个接收中断。

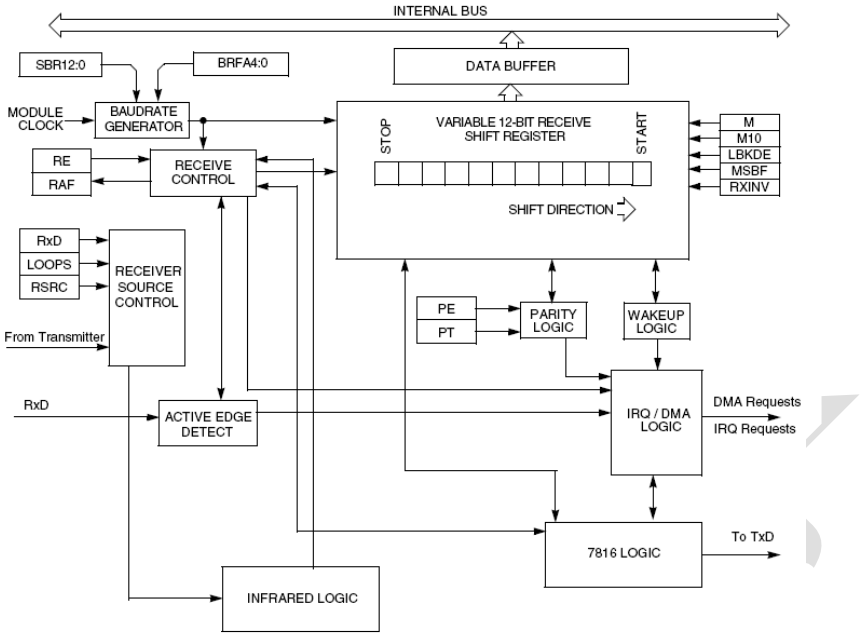

图 4-6 接收器的内部结构

4.3 MK60N512VMD100的UART模块的编程结构

表 4-3 给出了 MK60N512VMD100 的 UART 模块各寄存器相关信息，最左边一列给出了各寄存器的映射地址。最后三列分别给出了相应寄存器的位宽、访问权限和复位时的状态信息。表中给出的都是 UART0 的地址，对于其他 5 个 UART 来说，同名寄存器的地址相对于 UART0 的偏移分别为 0x0000_1000、0x0000_2000、0x0000_3000、0x0008_0000、0x0008_1000。例如，UART0 的波特率高字节寄存器的基址为 0x4006_A000，则 UART1 为 0x4006_B000，UART2 为 0x4006_C000，UART3 为 0x4006_D000，UART4 为 0x400E_A000，UART5 为 0x400E_B000，其他寄存器类似。

表 4-3 UART 模块存储映射

寄存器地址 UART0	寄存器	位宽	权限	复位值
0x4006_A000	UART波特率高字节寄存器（UARTx_BDH）	8	读/写	0x00
0x4006_A001	UART波特率低字节寄存器（UARTx_BDL）	8	读/写	0x04
0x4006_A002	UART控制寄存器1（UARTx_C1）	8	读/写	0x00
0x4006_A003	UART控制寄存器2（UARTx_C2）	8	读/写	0x00
0x4006_A004	UART状态寄存器1（UARTx_S1）	8	只读	0xC0
0x4006_A005	UART状态寄存器2（UARTx_S2）	8	读/写	0x00
0x4006_A006	UART控制寄存器3（UARTx_C3）	8	读/写	0x00
0x4006_A007	UART数据寄存器（(UARTx_D）	8	读/写	0x00
0x4006_A008	UART地址匹配寄存器1（UARTx_MA1）	8	读/写	0x00
0x4006_A009	UART地址匹配寄存器2（UARTx_MA2）	8	读/写	0x00
0x4006_A00A	UART控制寄存器4（UARTx_C4）	8	读/写	0x00
0x4006_A00B	UART控制寄存器5（UARTx_C5）	8	读/写	0x00
0x4006_A00C	UART扩展数据寄存器（UARTx_ED）	8	只读	0x00
0x4006_A00D	UART调制解调器寄存器（UARTx_MODEM）	8	读/写	0x00
0x4006_A00E	UART红外寄存器（UARTx_IR）	8	读/写	0x00

0x4006_A010	UART FIFO参数寄存器（UARTx_PFIFO）	8	读/写	0x00
0x4006_A011	UART FIFO控制寄存器（UARTx_CFIFO）	8	读/写	0x00
0x4006_A012	UART FIFO状态寄存器（UARTx_SFIFO）	8	读/写	0xC0
0x4006_A013	UART FIFO发送水位寄存器（UARTx_TWFIFO）	8	读/写	0x00
0x4006_A014	UART FIFO发送计数寄存器（UARTx_TCFIFO）	8	只读	0x00
0x4006_A015	UART FIFO接收水位寄存器（UARTx_RWFIFO）	8	读/写	0x01
0x4006_A016	UART FIFO接收计数寄存器（UARTx_RCFIFO）	8	只读	0x00
0x4006_A018	UART 7816控制数寄存器（UARTx_C7816）	8	读/写	0x00
0x4006_A019	UART 7816中断使能寄存器（UARTx_IE7816）	8	读/写	0x00
0x4006_A01A	UART 7816中断状态寄存器（UARTx_IS7816）	8	读/写	0x00
0x4006_A01B	UART 7816等待参数寄存器（UARTx_WP7816T0）	8	读/写	0x0A
0x4006_A01B	UART 7816等待参数寄存器（UARTx_WP7816T1）	8	读/写	0x0A
0x4006_A01C	UART 7816等待N寄存器（UARTx_WN7816）	8	读/写	0x00
0x4006_A01D	UART 7816等待FD寄存器（UARTx_WF7816）	8	读/写	0x01
0x4006_A01E	UART 7816错误阈值寄存器（UARTx_ET7816）	8	读/写	0x00
0x4006_A01F	UART 7816发送长度寄存器（UARTx_TL7816）	8	读/写	0x00

以上寄存器的用法在 MK60N512VMD100 的芯片手册上有详细的说明，下面按初始化顺序简要阐述基本编程需要使用的寄存器。注意，下面所列寄存器名中的“x”表示 UART 模块编号，取 0~5。

1．UART 控制寄存器 2（UARTx_C2）

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
复位	0							

D7—TIE，发送器中断或 DMA 传送使能。TIE 根据 C5[TDMAS]使能 S1[TDRE]标志产生中断请求或者 DMA 传送请求。0：TDRE 中断和 DMA 传送请求禁止。1：TDRE 中断或者 DMA 传送使能。

D6—TCIE，传送结束中断使能。TCIE 使能 S1[TC]传送完成标志产生中断请求。0：TC 中断请求禁止。1：TC 中断请求使能。

D5—RIE，接收器满中断或 DMA 传送使能。RIE 根据 C5[RDMAS]使能 S1[RDRF]标志产生中断请求或 DMA 传送请求。0：RDRF 中断和 DMA 传送请求禁止。1：RDRF 中断或 DMA 传送请求使能。

D4—ILIE，空闲线中断使能。ILIE 根据 C5[ILDMAS]的状态使能 S1[IDLE]空闲线标志产生中断请求。0：IDLE 中断请求禁止。1：IDLE 中断请求使能。

D3—TE，发送器使能。TE 使能 UART 发送器。TE 位可以通过清 0 然后置位 TE 位来排列一个空闲前导。当 7816E 被置位（使能）并且 C7816[TTYPE]=1 时，TE 位在请求块被发送之后会被自动清 0。当 TL7816[TLEN]=0 并且四个附加字符被发送时，TE 位自动清 0 的条件会一直被检测是否达到。0：发送器关闭。1：发送器开启。

D2—RE，接收器使能。RE 使能 UART 接收器。0：接收器关闭。1：接收器开启。

D1—RWU，接收器唤醒控制。RWU 可以被置位来使得 UART 接收器处于备用状态。当 RWU 事件（C1[WAKE]被清 0 的 IDLE 事件或者 C1[WAKE]被置位时的地址匹配）发生时，RWU 自动清 0。当 7816E 被置位时，此位必须被清 0。0：正常操作。1：RWU 使能唤醒功能并禁止接收器中断请求。通常，硬件通过自动清 0RWU 来唤醒接收器。

D0—SBK，发送中止。锁住的 SBK 发送一个中止字符（S2[BRK13]清 0 时，10、11 或 12 个逻辑 0；S2[BRK13]置位时，13 或 14 个逻辑 0）。锁住意味着在中止字符发送结束之前清除 SBK 位。只要 SBK 被置位，发送器会继续发送完整的中止字符。当 7816E 被置位时，此位必须被清 0。0：正常发送器操作。1：发送排列好的中止字符。

2．UART 控制寄存器 1（UARTx_C1）

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	LOOPS	UARTSWAI	RSRC	M	WAKE	ILT	PE	PT
复位	0							

D7—LOOPS，循环模式选择。当 LOOPS 被置位时，RxD 引脚从 UART 分离，发送器输出内部连接到接收器输入。发送器和接收器必须使能来使用循环功能。0：正常操作。1：发送器输出内部连接到接收器输入的循环模式。接收器输入由 RSRC 位决定。

D6—UARTSWAI，UART 在等待模式停止。0：在等待模式 UART 时钟继续运行。1：当 CPU 处于等待模式时，UART 时钟冻结。

D5—RSRC，接收器信号源选择。这个位在 LOOPS 位被置位时才有意义。当 LOOPS 被置位时，RSRC 位决定接收器移位寄存器输入的信号源。0：选择内部回环模式，接收器输入内部连接到发送器输出。1：单线 UART 模式，接收器输入连接到发送器引脚输入信号。

D4—M，9 位或 8 位模式选择。当 7816E 被置位（使能）时，此位必须被置位。0：正常模式-起始位+8 位数据位（由 MSBF 决定 MSB/LSB 优先）+停止位。1：使用模式-起始位+9 位数据位（由 MSBF 决定 MSB/LSB 优先）+停止位。

D3—WAKE，接收器唤醒方法选择。WAKE 决定哪种条件唤醒 UART：接收数据字符最高位的地址标记或者接收引脚输入信号上的空闲情况。0：空闲线唤醒。1：地址标记唤醒。

D2—ILT，空闲线类型选择。ILT 决定接收器何时开始计数当作空闲字符的逻辑 1。在一个有效的起始位或者停止位之后计数开始。如果起始位之后计数开始，那么停止位前的逻辑 1 的字符串可能导致空闲字符的错误识别。停止位后开始计数避免了错误的空闲字符识别，但是需要合适的同步传输。0：起始位后开始计数空闲字符位。1：停止位后开始计数空闲字符位。

D1—PE，奇偶校验使能。使能奇偶校验功能。当奇偶校验使能时，停止位前会被插入一个奇偶校验位。7816E 被置位（使能）时，此位必须被置位。0：奇偶校验功能禁止。1：奇偶校验功能使能。

D0—PT，校验类型。PT 决定了 UART 是否产生并检查奇校验位或者偶校验位。偶校验位中，偶数个 1 会清校验位，奇数个 1 会置位校验位。奇校验位中，奇数个 1 会清校验位，偶数个 1 会置位校验位。当 7816E 被置位（使能）时，此位必须清零。0：偶校验。1：奇校验。

3．UART 波特率高字节寄存器（UARTx_BDH）

UARTx_BDH 寄存器与 UARTx_BDL 寄存器一起控制 UART 波特率发生器的预分频因子。更新 13 位波特率设置（SBR[12:0]）时，首先写 BDH 缓冲新值的高半部分，然后写 BDL。直到 BDL 被写入时 BDH 中的值才会变。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	LBKDIE	RXEDGIE		SBR				
复位	0							

D7—LBKDIE，LIN 中止检测中断使能。LBKDIE 根据 LBKDDMAS 的状态使能 LIN 中止检测标识 LBKDIF 来产生中断请求。0：LBKDIF 中断请求禁止。1：LBKDIF 中断请求使能。

D6—RXEDGIE，RxD 输入有效边沿中断使能。RXEDGIE 使能接收输入有效边沿 RXEDGIF 来产生中断请求。0：RXEDGIF 硬件中断禁止（使用轮询）。1：RXEDGIF 中断请求使能。

D4~D0—SBR，UART 波特率位。UART 的波特率由这 5 位和 UARTx_BDL 共 13 位决定。UARTx_BDL 的复位值为 0x04。

4．UART 控制寄存器 4（UARTx_C4）

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	MAEN1	MAEN2	M10	BRFA				
复位	0							

D7—MAEN1，地址匹配模式使能 1。0：如果 MAEN2 清零，所有接收到的数据被传送到数据缓冲区中。1：所有最高有效位被清的接收数据被丢弃。所有最高有效位被置位的接收数据与 MA1 寄存器中的内容比较。如果没有匹配成功，则数据被丢弃。如果匹配，那么数据传送到数据缓冲区。当 C7816[ISO7816E]被置位（使能）时，此位必须清零。

D6—MAEN2，地址匹配模式使能 2。0：如果 MAEN1 清零，则所有接收的数据被传送到数据缓冲区。1：所有最高有效位清零的接收数据被丢弃。所有最高有效位置位的接收数据，跟 MA2 寄存器中的内容比较。如果没有匹配成功，数据被丢弃。如果匹配，数据被传送到数据缓冲区。当 C7816[ISO7816E]被置位（使能），此位必须清零。

D5—M10，10 位模式选择。M10 位使得第十位非存储器映射位到串行传输中。第十位为奇偶校验位。M10 位并不影响 LIN 发送或者检测中止。如果 M10 被置位，C1[M]和 C1[PE]必须被置位。当 C7816[ISO7816E]被置位（使能），此位必须清零。0：奇偶校验位是串行传输中的第 9 位。1：奇偶校验位是串行传输中的第 10 位。

D4~D0—BRFA，波特率微调。这个位用来对平均波特率以 1/32 的增量增加时间精度。

5．UART 状态寄存器 1（UARTx_S1）

UARTx_S1 寄存器为 UART 中断或 DMA 请求提供 MCU 的输入。这个寄存器也可以由 MCU 进行轮询来检测。可以通过读状态寄存器之后读或写（取决于中断标志类型）UART 数据寄存器来清除标志。其他的指令只要不影响 I/O 处理也可以插入到上述两步中执行。

数据位	D7	D6	D5	D4	D3	D2	D1	D0
定义	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
复位	1	1	0	0	0	0	0	0

D7—TDRE，发送数据寄存器空标志。当发送缓冲区（D 和 C3[T8]）中的数据字的数目等于或少于 TWFIFO[TXWATER]中的数目时，TDRE 会被置位。正在传输过程中的字符不包含在此计数中。TDRE 置位时读 S1，然后写 UART 数据寄存器（D）会清 TDRE。为了能够获得更有效的中断服务，除了写到缓冲区的最终值之外的所有数据应该都写入 D/C3[T8]。然后在写最终数据前读 S1 可以清 TRDE 标志。0：发送缓冲区中的数据数目比 TWFIFO[TXWATER]中的数目大。1：自从标志清零后，发送缓冲区中的数据数目等于或少于 TWFIFO[TXWATER]中的数目。

D6—TC，发送完成标志。当有效数据传输过程在进行或有前导符或中止符被加载时，TC 被清零。当发送缓冲区为空并且没有数据，也没有前导或者中止符正在传输时，TC 被置位。当 TC 为 1 时，发送数据输出信号变成空闲状态（逻辑 1）。当 7816E 被置位（使能）时，此位在任何一个 NACK 信号接收到了之后但在任何相应的保护期期满之前被置位。清零 TC，需要在 TC 为 1 时读 S1，或者将新的发送数据写入 UART 数据寄存器（D）或者通过清零 C2[TE]位后置位来排队一个前导或者置位 C2 的 SBK 来排队一个中止符。0：发送进行中（发送数据、前导符或者中止字符）。1：发送空闲中（发送完成）。

D5—RDRF，接收数据寄存器满标志。当接收缓冲区中数据字数目等于或多于 TWFIFO[TXWATER]中的数目时，RDRF 被置位。正在接收中的字符不包含在此计数中。当

S2[LBKDE]为 1 时，RDRF 只能为 0。另外，当 S2[LBKDE]为 1 时，接收的数据字会被存储在接收缓冲区中但会彼此覆盖。RDRF 为 1 时读 S1，然后读 UART 数据寄存器（D）会清 RDRF。为了获得更有效的中断和 DMA 操作，除了最终从缓冲区读出的数据之外的所有数据都使用 D/C3[T8]/ED。读 S1 和最终数据值会清 RDRF。0：接收缓冲区中数据字的数目少于 RXWATER 中的数目。1：此标志被清之后，接收缓冲区中数据字的数目等于或多于 RXWATER 中的数目。

D4—IDLE，空闲线标志。当 10 个连续的逻辑 1（如果 C1[M]=0）、11 个连续的逻辑 1（C1[M]=1、C4[M10]=0）或者 12 个连续的逻辑 1（C1[M]=1、C4[M10]=1、C1[PE]=1）出现在接收器输入时，IDLE 为 1。IDLE 标志清零后，必须接收一个帧（虽然 C2[RWU]为 1 时没有必要存储在数据缓冲中）或者一个 LIN 中止字符必须在一个空闲状态可以置位 IDLE 标志之前置位 S2[LBKDIF]标志。在 IDLE 为 1 时读 UART 状态 S1 然后读 D 可以清 IDLE。当 7816E 被置位（使能）时，空闲检测不被支持，因此这个标志被忽略。0：自从 IDLE 标志上一次清零后，接收器输入现在没有数据或者从来都没有数据。1：自从上一次被置位后，接收器输入变成空闲的或者一直都没有被清。

D3—OR，接收器溢出标志。当软件没有成功防止接收数据寄存器数据溢出时，OR 为 1。OR 位会在数据字溢出缓冲区，并收到停止位后被置位。此时，所有其他错误标志（FE、NF 和 PF）都不能被置位。移位寄存器中的数据丢失，但是已经在 UART 数据寄存器中的数据没有收到影响。如果 OR 标志被置位，即使有足够的空间也不会有数据存储在数据缓冲区中。OR 被置位时读 S1 然后读 UART 数据寄存器（D）会清 OR。在 7816 模式中，可以通过编程 C7816[ONACK]位配置返回的 NACK。0：自从上一次标志被清后没有溢出发生。1：自从上一次溢出发生后，溢出发生或者溢出标志没被清除。

D2—NF，噪声标志。UART 在接收器输入检测到噪声时，NF 被置位。在溢出或 LIN 中止检测功能使能（S2[LBKDE]=1）时，NF 不会被置位。NF 被置位表明自从它上次被清后接收到带有噪声的数据字。读 S1 然后读 UART 数据寄存器（D）可以清 NF。0：自从上次标志被清后没有检测到噪声。如果接收缓冲区深度大于 1，那么收器缓冲区的数据可能带有噪声。1：自从上次标志被清后至少有一个接收到的数据字带有噪声。

D1—FE，帧错误标志。当逻辑 0 被接收当作停止位时，FE 被置位。在溢出或 LIN 中止检测功能使能（S2[LBKDE]=1）时，FE 不会被置位。FE 阻止进一步的数据接收直到它被清零。FE 为 1 时读 S1 然后读 UART 数据寄存器（D）可以清 FE。数据缓冲区中最后一个数据代表了帧错误使能的接收数据。然而，当 7816E 被置位（使能）时，帧错误不被支持。0：没有检测到帧错误。1：帧错误。

D0—PF，奇偶校验错误标志。当 PE 被置位、S2[LBKDE]禁用并且接收到的数据不能和它的奇偶校验位匹配时，PF 被置位。在溢出条件下 PF 不会被置位。PF 位被置位仅表示自从它上次被清零后，接收到奇偶校验错误的数据字。读 S1 然后读 UART 数据寄存器（D），可以清 PF。0：自从上次这个标志清零后，没有检测到奇偶检验错误。如果接收缓冲区深度大于 1，那么接收缓冲区中有可能有数据带有奇偶校验错误。1：自从上次这个标志被清零后，至少接收一个带有奇偶校验错误的数据字。

6．UART 数据寄存器（UARTx_D）

UARTx_D 其实是两个单独的寄存器，读时会返回只读接收数据寄存器中的内容，写时会写到只写发送数据寄存器。

4.4 基于构件方法的UART编程

4.4.1 UART 构件的函数原型设计

UART 具有初始化、接收和发送三种基本操作。按照构件的思想，可将它们封装成三个独立的功能函数，初始化函数完成对 UART 模块的工作属性的设定，接收和发送功能函数则完成实际的通信任务。对 UART 模块进行编程，实际上已经涉及到对硬件底层寄存器的直接操作，因此，可将初始化、接收和发送三种基本操作所对应的功能函数共同放置在命名为 uart.c 的文件中，并按照相对严格的构件设计原则对其进行封装，同时配以命名为 uart.h 的头文件，用来定义模块的基本信息和对外接口。

按照模块所具有的基本操作来确定构件中应该具有哪些功能集合，是很自然也很重要的事情。但是，要实现编程的构件化，对具体的函数原型的设计则是重中之重。函数原型设计的好坏直接体现构件化编程的成败。下面就以 UART 的初始化、接收和发送三种基本操作为例，来说明实现构件化的全过程。

需要说明的是，实现构件化编程的 UART 软件模块应当具有以下几个特点：

（1）UART 模块是最底层的构件，它主要向上提供三种服务，分别是 UART 模块的初始化、接收单个字节和发送单个字节，向下则直接访问模块寄存器，实现对硬件的直接操作。另外，从现实使用角度出发，它还需要封装接收 N 个字节和发送 N 个字节的子功能函数。

（2）UART 模块在软件上对应 1 个 uart.c 程序源代码文件和 1 个 uart.h 头文件，当需要对它进行移植时，大多数情况下只需简单拷贝这两个文件即可，无需对源代码文件和头文件进行修改，只有当实施不同芯片之间的移植时，才需要修改头文件中与硬件相关的宏定义。

（3）上层构件或软件在使用该构件时，严格禁止通过全局变量来传递参数，所有的数据传递都直接通过函数的形式参数来接收。这样做不但使得接口简洁，更加避免了全局变量可能引发的安全隐患。

UART 模块是最基本最底层的构件，它在满足功能完整性的情况下，具有不可再分的特性，可以把它叫做“元构件”。下面简要介绍，按照构件化的思想对 UART 模块进行编程的过程。

通过以上分析，可以设计 UART 构件的 7 个基本功能函数。

（1）初始化

```
void uart_init(UART_MemMapPtr uartch, uint32 sysclk, uint32 baud);
```

（2）发送单个字节

```
void uart_send1(UART_MemMapPtr uartch, uint8 ch);
```

（3）接收单个字节

```
uint8 uart_re1(UART_MemMapPtr uartch,uint8 *ch);
```

（4）发送 N 个字节

```
void uart_sendN(UART_MemMapPtr uartch ,uint8* buff,uint16 len);
```

（5）接收 N 个字节

```
uint8 uart_reN(UART_MemMapPtr uartch ,uint8* buff,uint16 len);
```

（6）使能串口接收中断

```
void enableuartreint(UART_MemMapPtr uartch,uint8 irqno);
```

（7）禁止串口接收中断

```
void disableuartreint(UART_MemMapPtr uartch,uint8 irqno);
```

4.4.2 UART 构件的头文件

与 UART 通信子函数相关的文件有头文件 uart.h，以及包含 UART 初始化和收发子函数的程序文件 uart.c。

头文件uart.h中的内容可分为两个主要的部分，它们分别是7个函数原型的声明和外设模块寄存器相关信息的定义。前者给出了本UART构件对上层构件或软件所提供的接口函数，而后者则指明了本“元构件”与具体硬件相关的信息。串行通信头文件UART.h含有串行通信寄存器和标志位定义以及串行通信相关函数声明，下面简要给出了它的主要内容。以初始化函数uart_init为例，通道号、当前的系统时钟、希望实现的通信波特率以及初始化时是否使能中断，都被设计为函数参数。这样的话，应用程序和上层构件在使用（调用）它时，将具有极大的灵活性。文件还给出了必要的硬件相关信息，当要把该构件移植到其他芯片时，就必须检查并修改这些信息。

```
//-------------------------------------------------------------------------*
// 文件名: uart.h                                                          *
// 说  明: uart构件头文件                                                  *
//-------------------------------------------------------------------------*
#ifndef __UART_H__
#define __UART_H__
    //1 头文件
    #include "common.h"

    //2 宏定义
    //2.1 串口号宏定义
    #define UART0 UART0_BASE_PTR
    #define UART1 UART1_BASE_PTR
    #define UART2 UART2_BASE_PTR
    #define UART3 UART3_BASE_PTR
    #define UART4 UART4_BASE_PTR
    #define UART5 UART5_BASE_PTR

    //2.2 接收引脚irq号宏定义
    #define UART0irq 45
    #define UART1irq 47
    #define UART2irq 49
    #define UART3irq 51
    #define UART4irq 53
    #define UART5irq 55

    //3 函数声明

//-------------------------------------------------------------------------*
    //函数名: uart_init                                                     *
    //功  能: 初始化uartx模块。                                            *
    //参  数: uartch:串口号                                                 *
    //        sysclk:系统总线时钟,以MHz为单位                               *
    //        baud:波特率，如9600，38400等，一般来说，速度越慢，通信越稳     *
    //返  回: 无                                                            *
    //说  明:                                                               *

//-------------------------------------------------------------------------*
    void uart_init (UART_MemMapPtr uartch, uint32 sysclk, uint32 baud);


//-------------------------------------------------------------------------*
    //函数名: uart_rel                                                      *
    //功  能: 串行接受1个字节                                               *
```

```
    //参  数: uartch: 串口号                                                *
    //         ch:    接收到的字节                                          *
    //返  回: 成功:1;失败:0                                                *
    //说  明:                                                              *

  //---------------------------------------------------------------------*
    uint8 uart_re1 (UART_MemMapPtr uartch,uint8 *ch);


  //---------------------------------------------------------------------*
    //函数名: uart_send1                                                   *
    //功  能: 串行发送1个字节                                              *
    //参  数: uartch: 串口号                                               *
    //         ch:    要发送的字节                                         *
    //返  回: 无                                                           *
    //说  明:                                                              *

  //---------------------------------------------------------------------*
    void uart_send1 (UART_MemMapPtr uartch, uint8 ch);


  //---------------------------------------------------------------------*
    //函数名: uart_reN                                                     *
    //功  能: 串行 接收n个字节                                             *
    //参  数: uartch: 串口号                                               *
    //        buff: 接收缓冲区                                             *
    //        len:接收长度                                                 *
    //返  回: 1:成功;0:失败                                                *
    //说  明:                                                              *

  //---------------------------------------------------------------------*
    uint8  uart_reN (UART_MemMapPtr uartch ,uint8* buff,uint16 len);


  //---------------------------------------------------------------------*
    //函数名: uart_sendN                                                   *
    //功  能: 串行 接收n个字节                                             *
    //参  数: uartch: 串口号                                               *
    //        buff: 发送缓冲区                                             *
    //        len:发送长度                                                 *
    //返  回: 无                                                           *
    //说  明:                                                              *

  //---------------------------------------------------------------------*
    void uart_sendN (UART_MemMapPtr uartch ,uint8* buff,uint16 len);


  //---------------------------------------------------------------------*
    //函数名: enableuartreint                                              *
    //功  能: 开串口接收中断                                               *
    //参  数: uartch: 串口号                                               *
    //        irqno: 对应irq号                                             *
    //返  回: 无                                                           *
    //说  明:                                                              *

  //---------------------------------------------------------------------*
  void enableuartreint(UART_MemMapPtr uartch,uint8 irqno);
```

```
//-------------------------------------------------------------------------*
   //函数名: disableuartreint                                               *
   //功  能: 关串口接收中断                                                 *
   //参  数: uartch: 串口号                                                 *
   //        irqno: 对应irq号                                               *
   //返  回: 无                                                             *
   //说  明:                                                                *

//-------------------------------------------------------------------------*
   void disableuartreint(UART_MemMapPtr uartch,uint8 irqno);
#endif
```

4.4.3 UART 构件的源程序文件

1．UART 构件的初始化功能函数：uart_init

```
//-------------------------------------------------------------------------*
//函数名: uart_init                                                        *
//功  能: 初始化uartx模块。                                                *
//参  数: uartch:串口号                                                    *
//        sysclk:系统总线时钟，以MHz为单位                                 *
//        baud:波特率，如9600，38400等，一般来说，速度越慢，通信越稳       *
//返  回: 无                                                               *
//说  明:                                                                  *
//-------------------------------------------------------------------------*
void uart_init (UART_MemMapPtr uartch, uint32 sysclk, uint32 baud)
{
register uint16 sbr, brfa;
uint8 temp;

//使能引脚
if (uartch == UART0_BASE_PTR)
{
//在PTD6上使能UART0_TXD功能
PORTD_PCR6 = PORT_PCR_MUX(0x3)
//在PTD7上使能UART0_RXD
PORTD_PCR7 = PORT_PCR_MUX(0x3);
}
else if (uartch == UART1_BASE_PTR)
{
//在PTC4上使能UART1_TXD功能
PORTC_PCR4 = PORT_PCR_MUX(0x3);

//在PTC3上使能UART1_RXD
PORTC_PCR3 = PORT_PCR_MUX(0x3);
}
else if (uartch == UART2_BASE_PTR)
{
//在PTD3上使能UART2_TXD功能
PORTD_PCR3 = PORT_PCR_MUX(0x3);
//在PTD2上使能UART2_RXD
PORTD_PCR2 = PORT_PCR_MUX(0x3);
}
```

```
else if (uartch == UART3_BASE_PTR)
{
//在PTC17上使能UART3_TXD功能

PORTC_PCR17 = PORT_PCR_MUX(0x3);
//在PTC16上使能UART3_RXD
PORTC_PCR16 = PORT_PCR_MUX(0x3);
}
else if (uartch == UART4_BASE_PTR)
{
//在PTE24上使能UART4_TXD功能
PORTE_PCR24 = PORT_PCR_MUX(0x3);
//在PTE25上使能UART4_RXD
PORTE_PCR25 = PORT_PCR_MUX(0x3);
}
else if (uartch == UART5_BASE_PTR)
{
//在PTE8上使能UART5_TXD功能
PORTE_PCR8 = PORT_PCR_MUX(0x3);
//在PTE9上使能UART5_RXD
PORTE_PCR9 = PORT_PCR_MUX(0x3);
}

//使能串口时钟
if(uartch == UART0_BASE_PTR)
SIM_SCGC4 |= SIM_SCGC4_UART0_MASK;
Else if (uartch == UART1_BASE_PTR)
SIM_SCGC4 |= SIM_SCGC4_UART1_MASK;
Else if (uartch == UART2_BASE_PTR)
SIM_SCGC4 |= SIM_SCGC4_UART2_MASK;
Else if(uartch == UART3_BASE_PTR)
SIM_SCGC4 |= SIM_SCGC4_UART3_MASK;
Else if(uartch == UART4_BASE_PTR)
SIM_SCGC1 |= SIM_SCGC1_UART4_MASK;
else
SIM_SCGC1 |= SIM_SCGC1_UART5_MASK;


//禁止发送接受
UART_C2_REG(uartch) &= ~(UART_C2_TE_MASK

 UART_C2_RE_MASK );

//配置成8位无校验模式
UART_C1_REG(uartch) = 0;

//计算波特率，串口0、1使用内核时钟，其它串口使用外设时钟，系统时钟为
//外设时钟的2倍
if ((uartch == UART0_BASE_PTR) | (uartch == UART1_BASE_PTR))
sysclk+=sysclk;

sbr = (uint16)((sysclk*1000)/(baud * 16));
temp = UART_BDH_REG(uartch) & ~(UART_BDH_SBR(0x1F));
UART_BDH_REG(uartch) = temp |  UART_BDH_SBR(((sbr & 0x1F00) >> 8));
UART_BDL_REG(uartch) = (uint8)(sbr & UART_BDL_SBR_MASK);
brfa = (((sysclk*32000)/(baud * 16)) - (sbr * 32));
temp = UART_C4_REG(uartch) & ~(UART_C4_BRFA(0x1F));
```

```
    UART_C4_REG(uartch) = temp |  UART_C4_BRFA(brfa);

    //使能发送接受
    UART_C2_REG(uartch) |= (UART_C2_TE_MASK
                                                                    |
     UART_C2_RE_MASK );
    }
```

2．UART 构件单字节发送功能函数：uart_send1

```
    //-------------------------------------------------------------------------*
    //函数名: uart_send1                                                       *
    //功  能: 串行发送1个字节                                                  *
    //参  数: uartch: 串口号                                                   *
    //         ch:    要发送的字节                                             *
    //返  回: 无                                                               *
    //说  明:                                                                  *
    //-------------------------------------------------------------------------*
    void uart_send1 (UART_MemMapPtr uartch, uint8 ch)
    {
        //等待发送缓冲区空
        while(!(UART_S1_REG(uartch) & UART_S1_TDRE_MASK));
        //发送数据
        UART_D_REG(uartch) = (uint8)ch;
     }
```

3．UART 构件的单字节接收功能函数：uart_re1

```
//-------------------------------------------------------------------------*
//函数名: uart_re1                                                         *
//功  能: 串行接受1个字节                                                  *
//参  数: uartch: 串口号                                                   *
//         ch:    接收到的字节                                             *
//返  回: 成功:1;失败:0                                                    *
//说  明:                                                                  *
//-------------------------------------------------------------------------*
uint8 uart_re1 (UART_MemMapPtr uartch,uint8 *ch)
{
    uint32 k;

    for (k = 0; k < 0xfbbb; k++)//有时间限制
      if((UART_S1_REG(uartch) & UART_S1_RDRF_MASK)!= 0)//判断接收缓冲区是否满
      {
          *ch = UART_D_REG(uartch);
          return 1;                                                  //接受成功
      }
    if(k>=0xfbbb)
    {
        return 0;                                                    //接受失败
    }
    return 0;
}
```

4．UART 构件的多字节发送功能函数：uart_sendN

```
//-------------------------------------------------------------------------*
//函数名: uart_sendN                                                       *
//功  能: 串行 接收n个字节                                                 *
//参  数: uartch: 串口号                                                   *
//        buff: 发送缓冲区                                                 *
//        len:发送长度                                                     *
//返  回: 无                                                               *
//说  明:                                                                  *
//-------------------------------------------------------------------------*
void uart_sendN (UART_MemMapPtr uartch ,uint8* buff,uint16 len)
{
    int i;
    for(i=0;i<len;i++)
    {
uart_send1(uartch,buff[i]);
    }
}
```

5．UART 构件的多字节接收功能函数：uart_reN

```
//-------------------------------------------------------------------------*
//函数名: uart_reN                                                         *
//功  能: 串行 接收n个字节                                                 *
//参  数: uartch: 串口号                                                   *
//        buff: 接收缓冲区                                                 *
//        len:接收长度                                                     *
//返  回: 1:成功;0:失败                                                    *
//说  明:                                                                  *
//-------------------------------------------------------------------------*
uint8 uart_reN (UART_MemMapPtr uartch ,uint8* buff,uint16 len)
{
    uint16 m=0;
    while (m < len)
    {
        if(0==uart_re1(uartch,&buff[m]))
            return 0;  //接收失败
        else m++;
    }

    return 1;          //接收成功

}
```

6．UART 构件的使能串口接收中断功能函数：enableuartreint

```
//-------------------------------------------------------------------------*
//函数名: enableuartreint                                                  *
//功  能: 开串口接收中断                                                   *
//参  数: uartch: 串口号                                                   *
//        irqno: 对应irq号                                                 *
//返  回: 无                                                               *
//说  明:                                                                  *
//-------------------------------------------------------------------------*
void enableuartreint(UART_MemMapPtr uartch,uint8 irqno)
{
   UART_C2_REG(uartch)|=UART_C2_RIE_MASK;   //开放UART接收中断
enable_irq(irqno); //开接收引脚的IRQ中断
}
```

7．UART 构件的禁止串口接收中断功能函数：disableuartreint

```
//-------------------------------------------------------------------------*
//函数名: disableuartreint                                                 *
//功  能: 关串口接收中断                                                   *
//参  数: uartch: 串口号                                                   *
//        irqno: 对应irq号                                                 *
//返  回: 无                                                               *
//说  明:                                                                  *
//-------------------------------------------------------------------------*
void disableuartreint(UART_MemMapPtr uartch,uint8 irqno)
{
   UART_C2_REG(uartch)&=~UART_C2_RIE_MASK;   //禁止UART接收中断
   disable_irq(irqno);//关接收引脚的IRQ中断
}
```

4.4.4 UART 构件的测试工程

嵌入式软件中，main.c 和 isr.c 这两个文件反映了软件系统的整体执行流程。当系统启动并初始化后，程序根据 main.c 中定义的主循环顺序执行；一旦遇到中断请求，立即转去执行 isr.c 中定义的相应中断处理程序；当中断处理程序运行结束后，再返回中断处继续顺序执行。在 main.c 和 isr.c 中，可通过调用上一小节介绍的 7 个功能接口函数实现 UART 模块收发数据，实现方式有两种：查询方式和中断方式。

查询方式：UART3 模块首先向 PC 机发送字符串“Hello World!”，然后等待接收 PC 机从串口发送来的数据，若成功接收到 1 个数据，则立即将该数据回发给 PC 机，随后继续等待接收 1 个数据并回发，如此循环。主函数文件 main.c 如下编写：

```
//-----------------------------------------------------------------------*
// 工 程 名: uart_loop                                                    *
// 硬件连接: 将K60核心板与扩展板连接                                      *
// 程序描述: 启动后发送"Hello World!"，之后等待接收一个字节数据，收到后回发*
// 目    的: 初步掌握利用查询方式进行串行通信的基本知识                  *
// 说    明: 波特率为9600,使用UART3口                                    *
//---------------苏州大学飞思卡尔嵌入式系统实验室2011年-------------------*
//头文件
#include "includes.h"
//全局变量声明
extern int periph_clk_khz;
//主函数
void main(void)
{
    //1 主程序使用的变量定义
    uint32 runcount;                                    //运行计数器
    uint8 ch;
    //2 关中断
    DisableInterrupts;     //禁止总中断
    //3 模块初始化
    light_init(Light_Run_PORT,Light_Run1,Light_OFF); //指示灯初始化
    uart_init (UART3,periph_clk_khz,9600);                  //串口初始化
    //4 开中断

    uart_sendN(UART3,(uint8*)"Hello World!",12);
    //主循环
    while(1)
    {
        //1 主循环计数到一定的值，使小灯的亮、暗状态切换
        runcount++;
        if(runcount>=10)
        {
            light_change(Light_Run_PORT,Light_Run1);//指示灯的亮、暗状态切换
            runcount=0;
        }
        //2 串口接收一个字节的数据
        if(uart_reN(UART3,&ch,1))
        {
            uart_send1(UART3,ch);                           //发送回去
        }
    }
}
```

图 4-7 为测试串行通信的高端程序运行界面，建议读者充分使用串口工具。

图 4-7 串口调试工具软件界面

4.5 K60第一个带有中断功能的实例

采用中断方式收发数据时，需编写中断处理程序。在 CW 环境下使用 K60 芯片中断的步骤是：

（1）在 main.c 中，依照“关总中断→开模块中断→开总中断”的顺序打开模块中断；

（2）在 isr.c 文件中，编写中断服务程序；

（3）在 vectors.h 文件中，修改中断向量表；

K60 开始运行后，系统状态寄存器 SR 中 16、17 和 18 这三位的默认值都是“1”，即关闭所有中断，所以要使用中断，必须更改这三位的值。它就相当于一个总闸，如果总闸不开，所有中断都不可能发生。操作状态寄存器 SR，需要汇编指令来实现：

```
CPSIE i        //关总中断
CPSID i        //开总中断
```

为了方便代码移植，在 common.h 文件中做了如下定义：

```
#define EnableInterrupts asm(" CPSIE i");//开总中断
#define DisableInterrupts asm(" CPSID i");//关总中断
```

下面以 UART3 接收中断为例，实现以下功能：UART3 模块首先向 PC 机发送字符串“Hello World!”；同时，串口等待接收从 PC 机发来的数据，一旦接到数据，马上将该数据回发给 PC 机。串口接收程序使用中断来实现，中断处理程序执行完毕后，又回到主程序。

1．主函数文件（main.c）

```
//-----------------------------------------------------------------------*
// 工 程 名: uart_int                                                    *
// 硬件连接: 将K60核心板与扩展板连接                                     *
// 程序描述: 启动后发送"Hello World!"，之后等待接收一个字节数据，收到后回发*
// 目    的: 初步掌握利用中断方式进行串行通信的基本知识                  *
// 说    明: 波特率为9600,使用UART3口                                   *
//---------------苏州大学飞思卡尔嵌入式系统实验室2011年-------------------*
//头文件
#include "includes.h"
//全局变量声明
extern int periph_clk_khz;
//主函数
void main(void)
{
    //1 主程序使用的变量定义
    uint32 runcount;                                        //运行计数器
    //2 关中断
    DisableInterrupts;      //禁止总中断
    //3 模块初始化
    light_init(Light_Run_PORT,Light_Run1,Light_OFF); //指示灯初始化
    uart_init (UART3,periph_clk_khz,9600);                          //串口初
 始化
    //4 开中断
    enableuartreint(UART3,UART3irq);                        //开串口3接收
 中断
    EnableInterrupts;
 //开总中断

    uart_sendN(UART3,(uint8*)"Hello World!",12);
    //主循环
    while(1)
    {
        //1 主循环计数到一定的值，使小灯的亮、暗状态切换
        runcount++;
        if(runcount>=5000000)
        {
            light_change(Light_Run_PORT,Light_Run1);//指示灯的亮、暗状态切换
            runcount=0;
        }
    }
}
```

2．中断处理函数文件（isr.c）

以下简要给出了与本工程相关的中断函数定义。

```
//-------------------------------------------------------------------------*
// 文件名: isr.c                                                           *
// 说  明: 中断处理例程                                                    *
//---------------苏州大学飞思卡尔嵌入式系统实验室2011年--------------------*

#include "includes.h"

//-------------------------------------------------------------------------*
//函数名: uart3_isr                                                        *
//功  能: 串口3数据接收中断例程                                            *
//说  明: 无                                                               *
//-------------------------------------------------------------------------*
void uart3_isr(void)
{
    uint8 ch;
    DisableInterrupts;                                  //关总中断
    //接收一个字节数据并回发
 if(uart_re1 (UART3,&ch))
uart_send1(UART3,ch);
    EnableInterrupts;                                   //开总中断
}
```

3．中断向量表文件（vectors.s）

编写好中断处理函数后，接下来需要修改中断向量表。在 vectors.h 文件中，找到对应位置，把定义的中断服务程序名写入,然后再声明外部中断处理函数。以下简要给出了与本工程相关的中断向量表内容。

```
//-------------------------------------------------------------------------*
// 文件名: vectors.h                                                       *
// 说  明: 中断向量表宏定义                                                *
//-------------------------------------------------------------------------*
#ifndef __VECTORS_H
#define __VECTORS_H     1
    #include "common.h"
    //中断服务例程
    extern void uart3_isr(void);
    //默认中断服务例程
    void default_isr(void);
    void abort_isr(void);
    void hard_fault_handler_c(unsigned int * hardfault_args);

    typedef void pointer(void);

    extern void __startup(void);
    extern unsigned long __BOOT_STACK_ADDRESS[];
    extern void __iar_program_start(void);
    #define VECTOR_000      (pointer*)__BOOT_STACK_ADDRESS
#define VECTOR_001      __startup
…
#define VECTOR_067      uart3_isr
…
#endif
```

4.6 进一步讨论

本节将讨论串行通信的一些扩展应用。

4.6.1 流控制与 Break 信号

串口通信通常只用到 9 芯串口的 2（RXD）、3（TXD）和 5（GND）三个管脚。像这样的简单三线连接，使用的数据线少，通信中可节约通信成本，增加稳定性，能满足绝大多数的需求。但严格来说，当两个系统进行串行通信时，在接收方收完数据之前，应当禁止发送者发送新的数据，这个过程称为流控制（flow control）或握手（shake hands）。流控制常有软件方式和硬件方式。

软件握手又称为 XON/XOFF 方式，用于接受者和发送者之间无法实现硬件握手的场合。它用专门的字符来启动（XON）或停止（XOFF）数据流。这些字符定义在美国标准信息交换码（ASCII 码）中。软件握手用两个字符来实现流控制，一个代表请求对方暂停传输，另一个代表清除暂停的请求，继续数据传送。通常情况下用 ASCII 码为 0x13 和 0x11 二进制数据来表示，这样，在传输的数据中应禁止包含这两个字符，因为它们会被接收者理解为流控制字符而不是数据。假如发送的只是 ASCII 字符，就不会存在这样的问题，但发送二进制数据的话就会遇到这种情况。常见的解决办法是在发送前对二进制数据进行预处理，将它们用 ASCII 字符来代替。

在 RS-232 标准中，硬件握手使用 RTS（Request To Send，请求发送）和 CTS（Clear To Send，允许发送）两个信号。接收方当准备好接收数据时就将 CTS 信号设置成 0 电平，没有准备好就设置为 1 电平。同样，发送方当准备好发送数据时，就将 RTS 设置成 0 电平，没有准备好就将其设置为 1 电平。硬件流控制使用的是专门的信号设置，而软件要完成同样的任务需要发送和接收额外的数据，所以硬件方式比软件方式速度要快得多。

正常情况下一个数据接收或发送信号保持在高电平，直到一个新数据要传送。如果这个信号跳变到低电平并持续一个较长的时钟周期(通常是 1/4 到 1/2 秒)，便说明这是一个 Break（停顿）状态。它可以用来实现收发双方的同步，有时也用作复位或改变通信设备的操作模式，例如调制解调器。

4.6.2 延长串口通信的距离

串口使用一根发送信号线和一根接收信号线来构成共地的传输形式，这种共地传输容易产生共模干扰，抗噪性能弱。RS-232C 最大的传输距离大约是 30m，通信速率一般低于 20Kbps。当然，可以通过降低传输速率的方法来提升传输的距离，但除此之外，还可通过级联信号维持电路来解决这个问题。例如，MAX232 芯片具有两组 232 电平与 TTL 电平转换通路，在传输通路中，信号强度降低后，可先把 232 电平转化成 TTL 电平，再将 TTL 电平回转为 232 电平，这样获得的 232 电平信号又具有了强信号的特性，可保证传输更远的距离。从理论上来说，这种方法可以多次使用，传输距离想要多远就有多远，但这增加了系统开发的成本。

4.6.3 串口的扩展

通用 PC 的 COM 接口有 1 或 2 个，当 PC 机上需要挂载多个串口设备时，这显然不能满足需求，需要串口扩展。同样在以 MCU 为核心的多主机串口通信网络中，也极有可能出现一对多的串行通信方式。

目前比较通用的串口扩展方案有两种。一是用硬件实现，使用多串口单片机或专用串口

扩展芯片，可选的扩展芯片有 TI 等公司研发的 16C554 系列串口扩展芯片，该系列芯片通过并行口扩展串行口，功能比较强大、通讯速度高，但控制复杂，同时价格较高，主要的应用场合是 PC 机串口扩展产品。而多串口单片机也同样存在价格高的缺点。

另一种串口扩展方案就是用软件实现，当然，这种方案需要在硬件连接上使用总线式连接，软件上用广播式传输控制。具体来说，一个串口挂载多个串口设备，PC 机主动发送数据或控制信号到各个终端，终端则被动响应并适时回复。在终端回复的过程中，线路连接上需要考虑各个终端的信息干扰，可以在每个终端到回发线之间的连接一个单向传输的二极管，以解决此问题。软件模拟串口的缺点：一是采样次数低，分摊到每个终端设备的传输时间有限；二是不能实现高波特率通讯，一般不要使用高于 9600bps 的波特率，以保证传输正确性。

第5章 GPIO的应用实例——键盘、LED与LCD

5.1键盘技术概述

5.1.1 键盘模型及接口

5.1.2 键盘编程的基本问题

5.1.3 键盘构件设计与测试实例

5.2 LED技术概述

5.2.1 扫描法 LED 显示编程原理

5.2.2 LED 构件设计与测试实例

5.3 LCD技术概述

5.3.1 LCD 的特点和分类

5.3.2 点阵字符型液晶显示模块

5.3.3 HD44780

5.3.4 LCD 构件设计与测试实例

可参见芯片手册第 54 章通用输入输出来进行编程。

第6章 定时器相关模块

6.1计数器/定时器的基本工作原理

6.2 可编程延时模块PDB

6.2.1 PDB 工作原理

6.2.2 PDB 相关寄存器

6.2.3 PDB 构件设计及测试实例

涉及芯片手册 38 章（可编程延时寄存器）

6.3 Flex定时器FTM

6.3.1 FTM 工作原理

6.3.2 FTM 相关寄存器

6.3.3 FTM 中断

6.3.4 FTM 构件设计及测试实例

涉及芯片手册 39 章（Flex 定时器）

6.4 周期中断定时器PIT

6.4.1 PIT 工作原理

6.4.2 PIT 相关寄存器

6.4.3 PIT 构件设计及测试实例

涉及芯片手册 40 章（周期中断定时器）

6.5低功耗定时器LPTMR

6.5.1 LPTMR 工作原理

6.5.2 LPTMR 相关寄存器

6.5.3 LPTMR 构件设计及测试实例

涉及芯片手册 41 章（低功耗定时器）

6.6载波调制发射器CMT

6.6.1 CMT 工作原理

6.6.2 CMT 相关寄存器

6.6.3 CMT 中断

6.6.4 CMT 构件设计及测试实例

涉及芯片手册 42 章（载波调制发射器）

6.7实时时钟RTC

6.7.1 RTC 工作原理

6.7.2 RTC 相关寄存器

6.7.3 RTC 构件设计及测试实例

涉及芯片手册 26 章（RTC 振荡器）、43 章（实时时钟 RTC）

第7章 A/D与D/A

7.1 A/D和D/A转换的基本问题

7.2 A/D转换模块

7.2.1 A/D 转换模块寄存器

7.2.2 A/D 转换构件设计及测试实例

7.3 D/A转换模块

7.3.1 D/A 转换模块寄存器

7.3.2 D/A 转换构件设计及测试实例

7.4 比较器CMP概述

涉及第 34 章（AD 转换器）、35 章（比较器）、36 章（12 位 D/A 转换器）37 章（参考电压）

第8章 SPI

8.1 SPI的基本工作原理

8.1.1 SPI 概述

8.1.2 SPI 的数据传输

8.1.3 SPI 模块的时序

8.2 SPI模块的编程基础

8.2.1 SPI 模块的引脚

8.2.2 SPI 模块的寄存器

8.2.3 SPI 编程基本方法

8.3 SPI构件设计及测试实例

涉及芯片手册第 49 章

第9章 I2C与I2S

9.1 I2C总线概述

9.1.1 I2C 总线特点

9.1.2 I2C 总线标准的发展历史

9.1.3 I2C 总线的相关术语

9.2 I2C总线工作原理

9.2.1 总线上数据的有效性

9.2.2 总线上的信号

9.2.3 总线上数据的传输格式

9.2.4 总线寻址约定

9.3 I2C模块的编程基础

9.3.1 I2C 模块寄存器

9.3.2 I2C 模块编程基本方法

9.4 I2C构件设计及测试实例

9.5 I2S概述

9.5.1 I2S 的特点

9.5.2 I2S 操作模式

9.5.3 I2S 的相关寄存器定义

9.6 I2S的构件化设计与测试实例

涉及芯片手册第 50 章，53 章

第10章 Flash

10.5 Flash存储器的保护特性和安全性

10.5.1 周期性冗余检测 CRC

10.5.2 存储器映像密码加速单元 MMCAU

10.5.3 随机数操作 RNGB

涉及芯片手册第 27 章（FLASh 内存控制器）、28 章（Flash 存储器模块）、29 章（Flex 总线）、30 章（EzPort）、31 章（周期性冗余检测）、32 章（存储器映像密码加速单元 MMCAU）、33 章（随机数操作）

第11章 CAN模块FlexCAN

11.1 CAN总线通用知识

11.1.1 CAN 总线协议的历史概况

11.1.2 CAN 硬件系统的典型电路

11.1.3 CAN 总线的有关基本概念

11.1.4 帧结构

11.1.5 位时间

11.2 K60的CAN模块概述与编程结构

11.2.1 CAN 特性

11.2.2 操作模式

11.2.3 CAN 模块的内存映像及寄存器定义

11.2.4 CAN 报文缓冲区

11.3 K60的CAN模块报文发送与接收函数设计

11.3.1 数据帧发送/接收

11.3.2 远程帧发送与接收

11.3.3 仲裁处理、匹配处理及报文缓冲区管理

11.4 K60的CAN模块编程实例

11.4.1 初始化函数设计

11.4.2 K60 的 CAN 模块构件化设计及测试实例

涉及芯片手册 第 48 章

第12章 USB 2.0 编程

12.1 USB基本概念及硬件特性

12.1.1 USB 特性

12.1.2 USB 相关基本概念

12.1.3 USB 的物理特性

12.2 USB的通信协议

12.2.1 USB 基本通信单元：包

12.2.2 USB 通信中的事务处理

12.2.3 从设备的枚举看 USB 数据传输

12.3 K60 的USB模块功能简介

12.3.1 K60 的 USB 模块功能简介

12.3.2 K60 的 USB 模块主要寄存器介绍

12.4 K60 作为USB从机的开发方法

12.4.1 PC 端 USB 设备驱动程序的选择及基本原理

12.4.2 PC 作为 USB 主机的程序设计

12.4.3 K60 作为 USB 从机的程序设计

12.5 K60 作为USB主机的开发方法

12.5.1 K60 作为 USB 主机的基本功能

12.5.2 USB 主机与 USB 设备通信

12.6 K60的USB设备电量检测模块USBDCD

12.6.1 USBDCD 概述

12.6.2 USBDCD 的内存映射与寄存器定义

12.6.3 USBDCD 构件化设计与测试实例

12.7 K60的UAB电压调节器

12.7.1 电压调节器特征

12.7.2 电压调节器操作模式

涉及芯片手册第 45 章（USB 总线 OTG 控制器）、46 章（USB 设备电量检测模块）、47 章（USB 电压调节器）

第13章 高容量SD存储卡SDHC

13.1 SDHC基本概念及硬件特性

13.1.1 SD 概述

13.1.2 SD 相关基本概念

13.1.3 SD 的物理特性

13.2 SD的通信协议

13.2.1 SD 基本通信单元

13.2.2 SD 通信中的事务处理

13.2.3 SD 数据传输

13.3 K60的SD模块基本编程方法

13.3.1 K60 的 SD 模块功能简介

13.3.2 K60 的 SD 模块存储器映像与寄存器定义

13.3.3 K60 的 SD 模块构件化设计与测试实例

涉及芯片手册第 52 章

第14章 TSI

14.1 TSI概述

14.1.1 TSI 特点

14.1.2 TSI 的操作模式

14.1.3 TSI 信号描述

14.2 TSI编程

14.2.1 TSI 的相关寄存器定义

14.2.2 TSI 的功能描述

14.2.3 TSI 的构件化设计与测试实例

涉及芯片手册第 55 章

第15章 基于K60的嵌入式以太网

15.1 嵌入式以太网相关基础知识

15.1.1 以太网的由来与协议模型

15.1.2 以太网中主要物理设备

15.1.3 IEEE 1588 概述

15.1.4 相关名词解释

15.2 K60以太网概述

15.2.1 K60 以太网特性

15.2.2 K60 以太网外部引脚说明

15.2.3 K60 以太网存储映像与寄存器带那个一

15.3 链路层编程

15.3.1 MAC 帧格式

15.3.2 MAC 帧的接收与发送

15.3.3 MAC 帧收发测试实例

15.4 网络层及更高层编程

15.4.1 Ipv4 与 Ipv6 简介

15.4.2 ICMP 简介

15.4.3 UDP 简介

15.4.4 TCP 简介

15.4.5 测试实例

15.5 FIFO

15.5.1 FIFO 概述

15.5.2 FIFO 的接收与发送

15.5.3 FIFO 的保护机制

15.5.4 FIFO 测试实例

15.6 PHY管理接口与以太网接口

15.6.1 MDIO 简介

15.6.2 以太网接口的发送与接收

15.7 K60以太网模块的其他功能

15.7.1 全双工流控制操作

15.7.2 魔术包检测

15.7.3 IP 加速器控制

15.7.4 复位与停止控制

15.7.5 遗留缓冲区描述符

15.7.6 增强缓冲区描述符

涉及芯片手册第 44 章

第16章 系统时钟与其他功能模块

16.1 时钟模块

涉及第 3 章 3.4 节及第 5 章、第 24 章、25 章的内容。

16.2 芯片配置模块

16.2.1 芯片配置模块简介

16.2.2 芯片配置模块寄存器定义

涉及第 12 章内容

16.3 电源管理模块

16.3.1 电源模式

涉及第 7 章的内容

16.3.2 低功耗模式

涉及第 14 章、15 章的内容

16.4 端口控制与中断模块

涉及第 11 章内容

16.5 复位与启动模块

涉及第 6 章的内容

16.6 杂项控制模块

涉及第 16 章的内容

16.7 交叉开关模块

涉及第 17 章内容

16.8 看门狗

涉及第 22 章、23 章的内容

第17章 操作系统的移植

讲述操作系统的基本知识