

嵌入式系统

An Introduction to Embedded System

第十课、嵌入式系统的调试

教师：蔡铭

cm@zju.edu.cn

浙江大学计算机学院人工智能研究所
航天科技—浙江大学基础软件研发中心

关于bug

□ bug用以指代昆虫及节肢动物。



□ 在自然界里bug是人类的主要竞争者。

□ 圣经“十灾”的故事

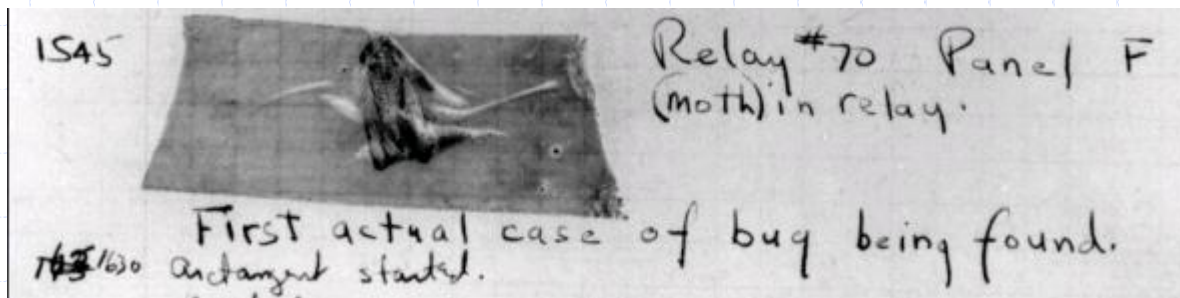
- 其中三种是由bug（虱灾、蝇灾、蝗灾）所引起的灾难



圣经故事

软件系统bug/debug的起源

- Grace Hopper (COBOL语言的发明者), 在 “Annals of the History of Computing “ (Vol. 3, No. 3 (July 1981), p285—286) 一书中, 记载了这段历史……
 - 1947年9月9日, 一个小虫子飞进了正在运行的Harvard Mark II计算机, 并导致该计算机死机。
 - Hopper带领她的小组检查哪里出了问题, 最后发现在第70号继电器这里出错。Hopper发现一只飞蛾躺在继电器里, 已经被继电器打死。她小心地用镊子将蛾子夹出来, 用透明胶布帖到“事件记录本”中, 并注明“第一个发现虫子的实例”。



the first bug - 1947.9.9



Quotes About Debug



- I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs——*Maurice Wilkes* (1967年图灵奖)



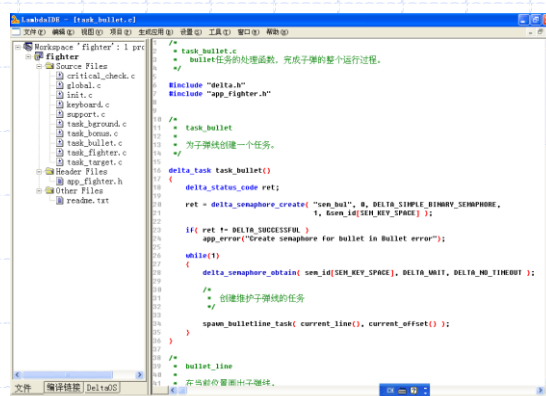
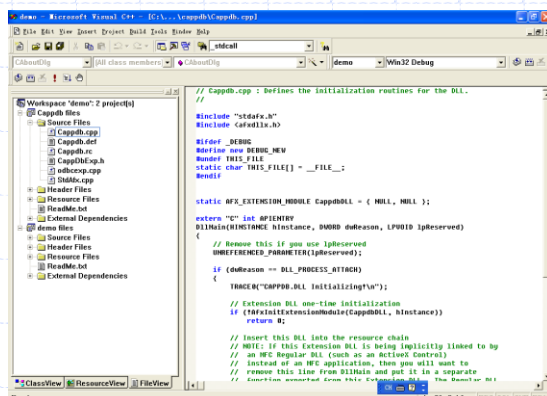
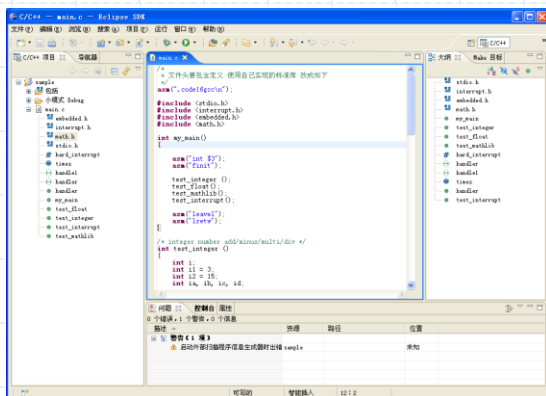
- If debugging is the process of removing bugs, then programming must be the process of putting them in——*Edsger W. Dijkstra* (1972年图灵奖)



- Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are—by definition—not smart enough to debug it——*Brian Kernighan*

软件调试的重要性

- ❑ 目前软件系统有如此多缺陷和故障的原因很多，其中一个重要因素是：
 - **软件调试**是一项**系统化**和**蕴涵丰富经验**的技术，尤其对于大型、复杂的软件系统而言，如何制定调试策略、进行故障诊断均很有技巧，而目前调试主要以手工方式为主进行，调试支持工具相对较弱，大量的程序员不擅长调试程序。



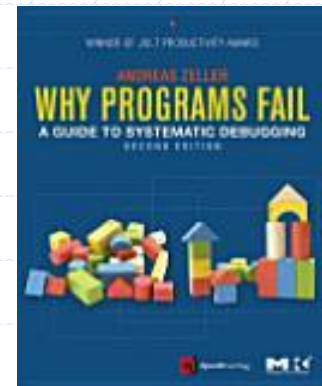
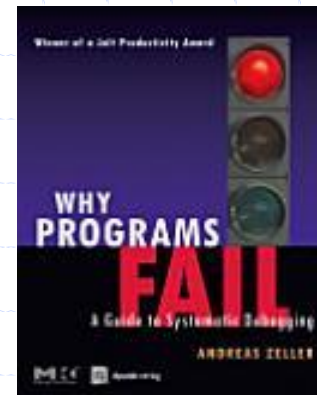
软件调试的基本步骤

□ 软件调试的TRAFFIC原则

- **T**rack the problem
- **R**eproduce the failure
- **A**utomate and simplify the test case
- **F**ind possible infection origins
- **F**ocus on the most likely origins
- **I**solate the infection chain
- **C**orrect the defect



Saarland
Prof.
Andreas
Zeller



bug种类繁多

□ bug分类体系：ANSI/IEEE 1044-1994、QJ3026-1998

■ 内存相关错误

■ 初始化错误

■ 计算错误

■ 输入输出错误

■ 控制流错误

■ 数据处理解释错误

■ 竞争类错误

■ 操作系统、体系结构、
编译器相关错误

■ 其他错误

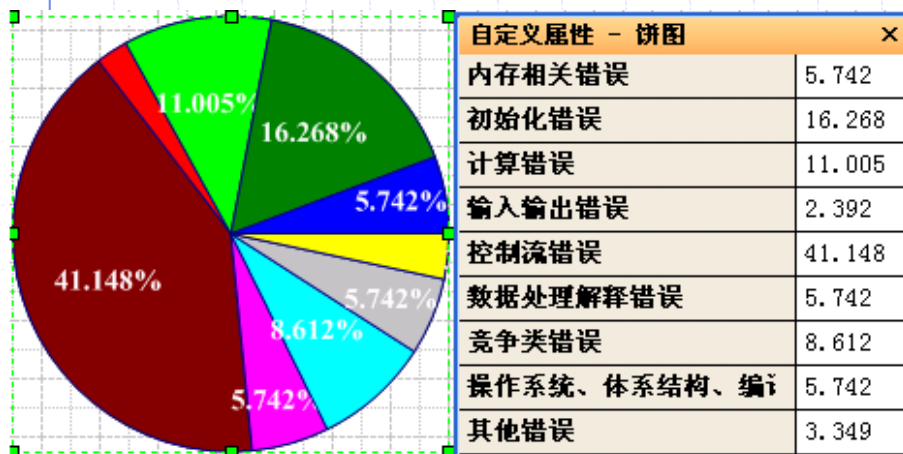
● 堆栈溢出 ● 多次释放
● 缓冲区溢出 ● 内存耗尽
● 内存泄漏

● 条件逻辑错误 ● 循环边界错误
● 条件比较错误 ● 函数调用错误
● 执行顺序错误 ● 状态转化错误

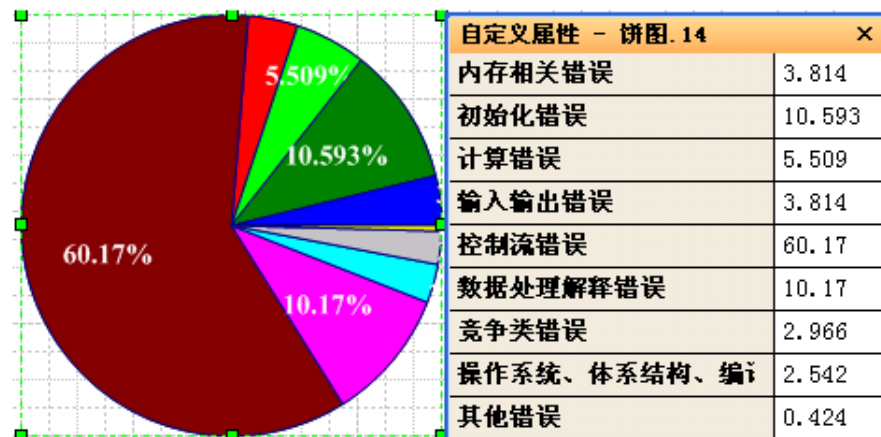
● 字节对齐错误 ● 宏扩展错误
● 浮点精度损失 ● 本地化错误
● 内存泄漏 ● 权限错误

bug分布特点实例分析

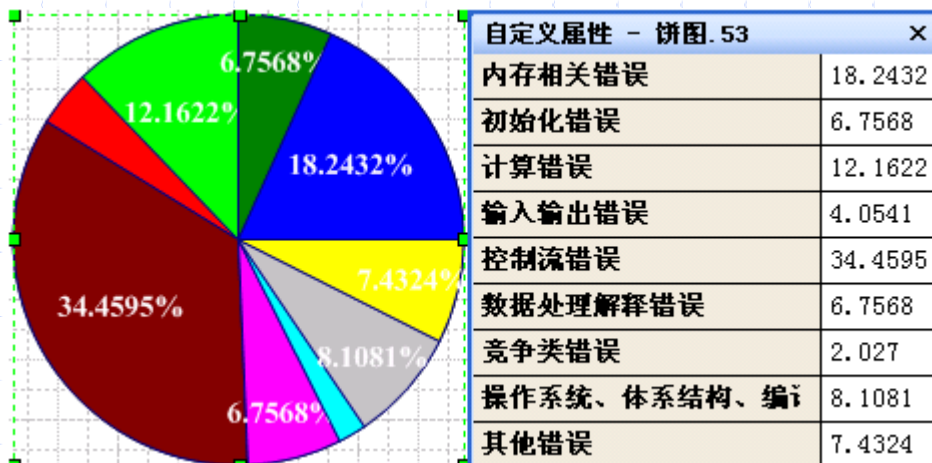
通过对RTEMS、Apache HTTPD、PostgreSQL等大型开源软件系统7年多来的近1000个bug分析，bug分布如下：



RTEMS BUG分布



HTTPD BUG分布



PostgreSQL BUG分布

软件调试 VS 软件测试

□ 软件测试与软件调试的区别

- ✓ 软件测试的目的是找出软件实现中与预定义的规格不符合的问题，即错误；
- ✓ 软件调试的目的是定位错误，并进行修改；
- ✓ 软件测试中发现的错误，需要通过调试来进行定位。

□ 软件调试的一般过程

- ✓ 观察故障现象
- ✓ 推断故障假设
- ✓ 确定故障位置
- ✓ 修正故障代码

上述四个步骤不断迭代进行，直至消除故障。其中，根据故障现象进行故障定位，是软件调试的关键环节。

课程大纲

 软件调试原理概述

 嵌入式软件调试概述

 GDB Server分析

 KGDB简介

软件调试器基本功能

单步执行
step in
step out

全局变量
局部变量
寄存器查看

The screenshot shows the Eclipse IDE with the GDB debugger. The top toolbar has a red circle around the 'step in' button (a magnifying glass over a right arrow). The 'Variables' tab on the right shows a list of variables and their values. The bottom panel shows the source code of 'main.c' and the assembly code being executed.

名称	值
cs	0
times	0
f1	0.5
f2	1.5
fa	1.58456325E29
fb	0.0
fc	0.0
fd	0.0

```
/* float number add/minus/multi/div */
test_float ()
{
    float f1 = 0.5;
    float f2 = 1.5;

    float fa = f1 + f2;
    float fb = f1 - f2;
    float fc = f1 * f2;
    float fd = f1 / f2;
}
```

```
0x000000c7 <test_float+9>: mov     $0x3f000000,%eax
0x000000cd <test_float+15>: addr32 mov %eax,0xffffffff
                        float f2 = 1.5;
0x000000d2 <test_float+20>: mov     $0x3fc00000,%eax
0x000000d8 <test_float+26>: addr32 mov %eax,0xffffffff

                        float fa = f1 + f2;
0x000000dd <test_float+31>: addr32 flds 0xffffffffc(%
0x000000e1 <test_float+35>: addr32 fadds 0xffffffff8(
0x000000e5 <test_float+39>: addr32 fstps 0xffffffff4(
                        f1... f2 = f1 / f2;
```

设置断点

软件调试的基本方法

□ 软件调试基本方法分类

- 操纵行为：操作被调试目标行为
- 访问数据：访问被调试目标数据
- 输出信息：让调试目标主动输出信息



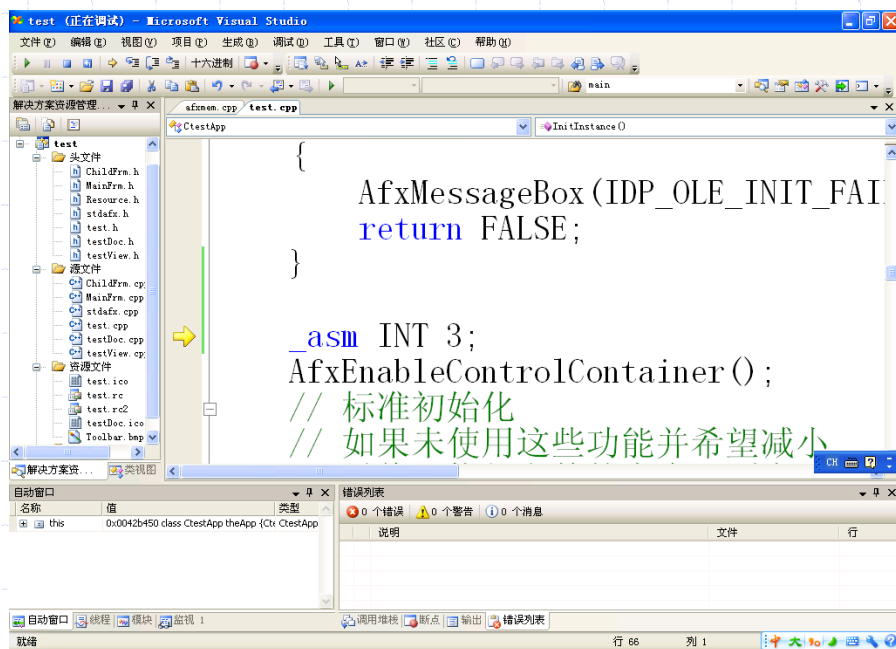
关于软件调试原理的小实验

□ 用VS创建一个基于MFC的工程项目

- 加入 `_asm INT 3`, 观察一下现象☺
- 看下 `_asm INT 3`的机器代码

□ 未分配空间的指针地址是什么?

能否发现一些关联关系?



关于软件调试器的几个疑问（1/2）

□ 关于软件调试的几个疑问

- ✓ 为什么软件要分为调试版（Debug）与发行版（Release），它们两者间的差异何在？
- ✓ 源代码级别的软件断点是如何设置，并与物理地址对应？

```
static void
decode_line_numbers (char *linetable)
{
    char *tblscan;
    char *tblend;
    unsigned long length;
    unsigned long base;
    unsigned long line;
    unsigned long pc;

    if (linetable != NULL)
    {
        tblscan = tblend = linetable;
        length = target_to_host (tblscan, sizeof LINETBL_LENGTH, GET_UNSIGNED,
                                current_objfile);
        tblscan += sizeof LINETBL_LENGTH;
        tblend += length;
        base = target_to_host (tblscan, TARGET_FT_POINTER_SIZE (objfile),
                               GET_UNSIGNED, current_objfile);
        tblscan += TARGET_FT_POINTER_SIZE (objfile);
        base += baseaddr;
```

0x400984

0x400A32

0x400AB8

0x400AE0

编译后目标代码

源程序： Dwarfread.c

关于软件调试器的几个疑问（2/2）

□ 关于软件调试的几个疑问

- ✓ 调试器与被调试软件是如何关联的？
- ✓ 调试指令（单步、step in、step out等）是如何实现的？
- ✓ 被调试软件的全局变量、局部变量、寄存器信息是如何查看的？

支持软件调试的几个方面

□ 编译器的支持

- 调试信息的注入

□ 计算机体系结构的支持

- 调试指令的支持
- 调试寄存器的支持

编译器的支持—调试版 (Debug) vs 发行版 (Release)

- 以elf-i386格式为例，DWARF (Debugging With Attributed Record Formats) 标准的调试版、发行版差异可通过objdump -h (列出表头) 进行比较
- 发行版: 22个节(section)
- 调试版: 29个节, 增加7个节包括:
 - ✓ .debug_arranges
 - ✓ .debug_pubnames
 - ✓ .debug_info
 - ✓ .debug_abbrev
 - ✓ .debug_line
 - ✓ .debug_frame
 - ✓ .debug_str



编译器的支持—调试版中的符号信息

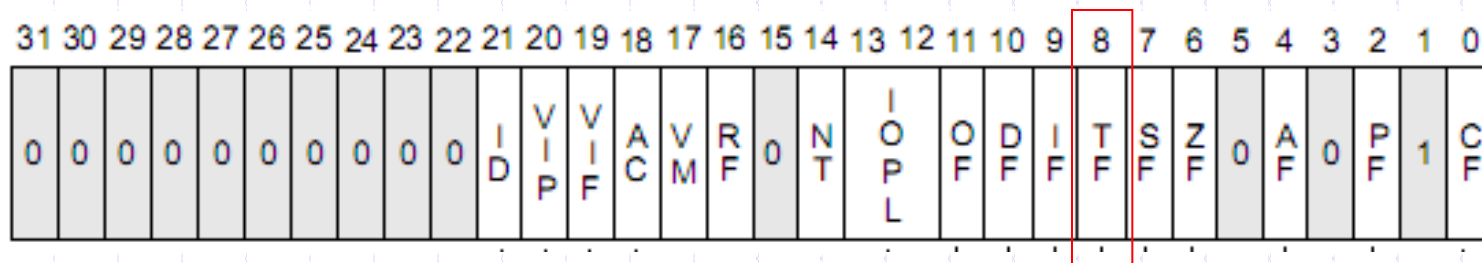
- 为实现源码级调试，编译、汇编、链接系统时，目标代码文件中产生了大量的关于目标代码与源代码之间联系的信息，包括：
 - 源代码行号与目标地址的对应关系表（.debug_line）
 - ◆ 源文件名：行号→目标地址
 - 全局变量、静态变量、函数入口地址对应关系表（.debug_pubnames）
 - ◆ 全局变量、静态变量、函数名→目标地址
 - 局部变量在堆栈地址偏移的对应关系表

体系结构对软件调试的支持

□ Intel体系结构对软件调试的支持

■ 断点指令：INT 3

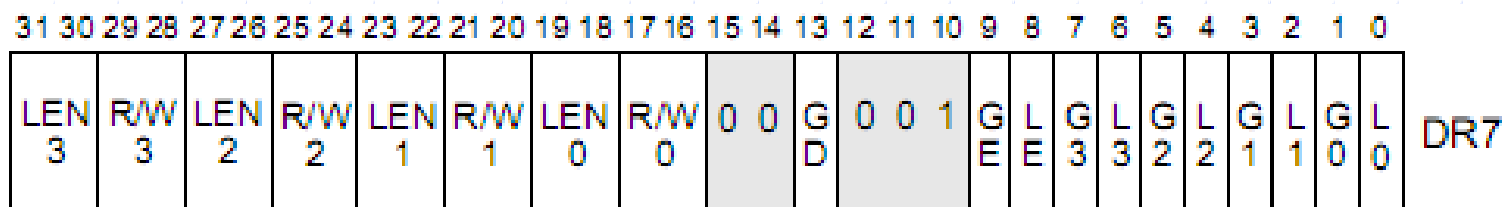
■ 状态寄存器（陷阱标志）：TF



■ 调试地址寄存器：DR0~DR3

■ 调试控制寄存器：DR7

■ 调试状态寄存器：DR6



软件调试中的断点分类

□ 断点的主要分类

- ✓ 代码断点：指令断点
- ✓ 数据断点：如x86中的DR0—DR3
- ✓ RAM断点：可读写断点
- ✓ ROM断点：只读断点

□ 通常意义的断点是指：RAM型代码断点

□ 嵌入式系统调试中的常用断点：ROM型代码断点

软件调试器是如何工作的？—断点设置

□ RAM型代码断点如何设置？

断点的设置	
输入	源文件的文件名和行号。
输出	物理地址或出错代码。
过程	根据源文件名和行号查询符号表，得到物理地址(如果无效，报错)。
	根据现有信息创建逻辑断点。
	根据物理地址搜索物理断点链： <ul style="list-style-type: none">● 如果存在，增加引用计数。● 如果不存在，创建物理断点，在目标代码中相应位置插入断点指令，并将原指令保存在物理断点中。

软件调试器是如何工作的？—异常处理

□ 调试器如何接管被调试软件？

■ 断点物理地址确定后，需插入调试异常指令，如：

◆ I386: int 3、ARM: BKPT

■ 调试器接管被调试软件方式

◆ 操作系统接管调试异常，并转交调试器

◆ 调试器直接接管调试异常

```
static void
read_lexical_block_scope (struct dieinfo *dip, char *thisdie, char *enddie,
                          struct objfile *objfile)
{
    register struct context_stack *new;

    push_context (0, dip->at_low_pc);
    process_dies (thisdie + dip->die_length, enddie, objfile);
    new = pop_context ();
    if (local_symbols != NULL)
    {
        finish_block (0, &local_symbols, new->old_blocks, new->start_addr,
                     dip->at_high_pc, objfile);
    }
    local_symbols = new->locals;
}
```

.debug_line

0xA0

0xA03248

0x0483

0x400AB8

0x400AE0

触发异常

0xCC3248

0x0483

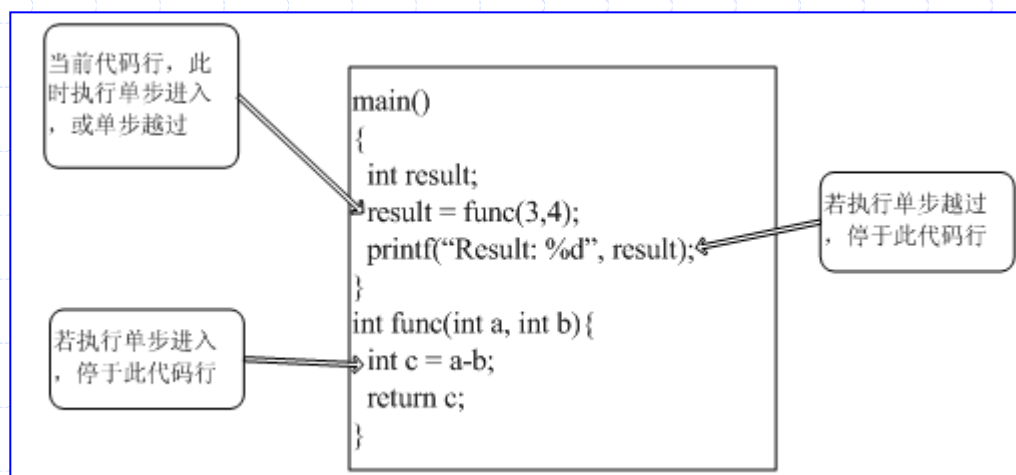
0x400AB8

0x400AE0

软件调试器是如何工作的？—单步执行

□ 单步执行

- 单步执行在调试行为中，具备非常重要的作用，调试者通过单步执行观测程序执行状态。
- 在程序出现异常时，通过单步执行，分析程序出现异常的过程和原因。
- 单步执行存在多种形式，如：单步进入、单步越过等。



软件调试器是如何工作的？—单步执行

□ 单步越过执行算法

1. `old_pc = cur_pc;`

2. 得到并且解析 `cur_pc` 指向的机器指令。

3. 如果该指令并不是跳转型指令，并且不对应某源码行起始点，使得 `cur_pc` 指向下一条机器指令，返回 2。

4. 如果该指令是函数调用指令(`call`)，在函数返回地址设置临时断点，继续执行程序直到停止，清除临时断点，设置 `cur_pc` 为当前机器指令地址。返回 2。

5. 如果该指令对应于某源码行起始点，

- `cur_pc != old_pc`
在该指令处设置临时断点，继续执行程序直到停止，清除临时断点，返回当前机器指令地址以及对应的源码行。
- `cur_pc = old_pc`
使得 `cur_pc` 指向下一条机器指令，返回 2。

6. 如果该指令是跳转指令，解析跳转指令，在所有可能跳转出口设置临时断点，继续执行程序直到停止，清除临时断点，设置 `cur_pc` 为当前机器指令地址，返回 2。

软件调试器是如何工作的？—单步执行

□ 单步进入执行算法

1. $old_pc = cur_pc$;

2. 得到并且解析 cur_pc 指向的机器指令。

3. 如果该指令并不是跳转型指令，并且不对应某源码行起始点，使得 cur_pc 指向下一条机器指令，返回 2。

4. 如果该指令对应于某源码行起始点，

- $cur_pc \neq old_pc$
在该指令处设置临时断点，继续执行程序直到停止，清除临时断点，返回当前机器指令地址以及对应的源码行。
- $cur_pc = old_pc$
使得 cur_pc 指向下一条机器指令，返回 2。

5. 如果该指令是跳转型指令，解析跳转指令，在所有可能跳转入口设置临时断点，继续执行程序直到停止，清除临时断点，设置 cur_pc 为当前机器指令地址，返回 2。









软件调试器是如何工作的？ — 信息查看

□ 被调试软件的信息查看

- ✓ 全局、静态变量：符号表debug_pubnames
- ✓ 寄存器信息：被调试任务上下文
- ✓ 局部变量：被调试任务上下文+符号表debug_frame

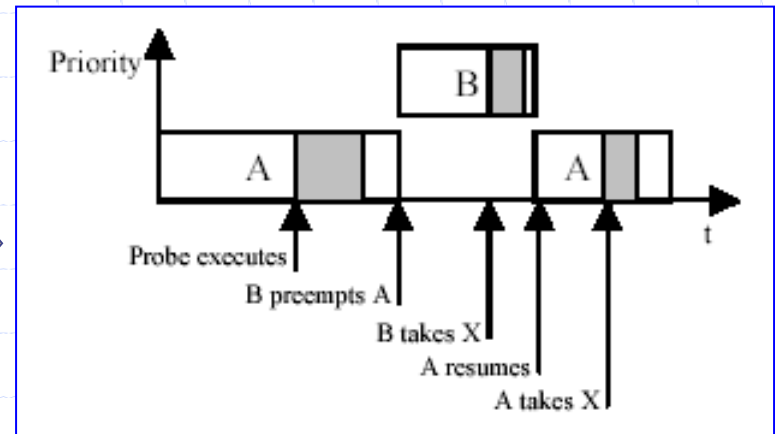
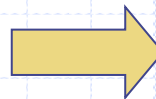
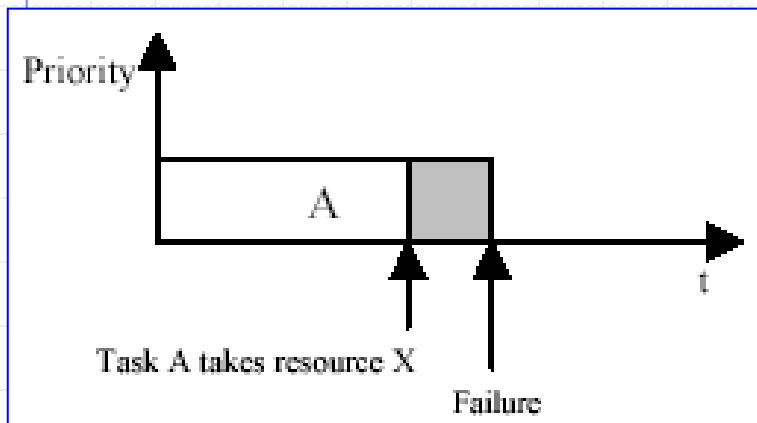
哪些软件难以调试？

- ❑ **系统底层软件**：如系统初始化程序（bootloader）、底层软件（如设备驱动程序）、复杂软件核心（如操作系统内核）等；
- ❑ **非确定性软件**：如多任务并发程序、实时应用系统、分布式系统等。

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.824±0.009	0	1585	1585	files[filesindex].language > 16
	0.176	0.824±0.009	0	1584	1584	strcmp > 0
	0.176	0.824±0.009	0	1580	1580	strcmp == 0
	0.176	0.824±0.009	0	1577	1577	files[filesindex].language == 17
	0.176	0.824±0.009	0	1576	1576	tmp == 0 is TRUE
	0.176	0.824±0.009	0	1573	1573	strcmp > 0
	0.116	0.883±0.012	1	774	775	((*(fi + i)))->this.last_line == 1
	0.116	0.883±0.012	1	776	777	((*(fi + i)))->other.last_line == yyleng
..... 2732 additional predictors follow						

当前软件调试器的局限性

- ❑ **干涉软件运行**：常用的软件调试器需要对软件系统进行插桩处理（源代码插桩或二进制代码插桩），改变了程序原有行为特征，将导致探针效应（**probe effect**）；



- ❑ **调试资源欠丰富**：例如，x86系统中仅支持4个数据断点（**DR0—DR3**），且数据断点跨度范围小（4字节）。

GNU调试器GDB简介（1/5）

□ GDB（Gnu Debugger）是GNU C自带的调试工具，使用GDB可以完成下面这些任务：

- ✓ 支持a.out、coff、ecoff、xcoff、elf、pe格式；
- ✓ 支持条件断点（如循环次数、条件表达式）设置；
- ✓ 提供运行状态查看（变量、寄存器）；
- ✓ 提供调试阶段的数据修改功能。

GNU调试器GDB简介（2/5）

□ GDB命令

- ✓ 键入gdb gdb_test命令来启动GDB并载入程序 gdb_test, 命令行进入了GDB模式。
- ✓ 命令补齐功能。

```
$ gdb gdb_test
```

```
.....
```

```
(gdb)
```

GNU调试器GDB简介（3/5）

□ GDB中的常用命令如下：

指令	说明
file	载入程序。如file hello。
quit	退出GDB。也可以输入'C-d'来退出GDB。
run	执行载入后的要调试的程序。可以输入参数。
info	查看程序的信息。多用来查看断点信息。可以用help info来查看具体帮助。 info source查看当前文件的名称，路径，所使用的程序语言等信息。 info stack 查看调用栈。 info local 查看局部变量信息。 info br br是断点break的缩写，用这条指令，可以得到所设置的所有断点的详细信息。
list	list FUNCTION列出被调试程序某个函数 list LINENUM以当前源文件的某行为中间显示一段源程序 list 接着前一次继续显示 list - 显示前一次之前的源程序 list FILENAME:FUNCTION显示另一个文件的一段程序，

GNU调试器GDB简介（4/5）

break	<p>最常用和最重要的命令：设置断点。break FUNCTION在函数入口设置断点 break LINENUM在当前源文件的某一行上设置断点 break FILENAME:LINENUM在另一个源文件的某一行上设置断点 break *ADDRESS在某个地址上设置断点</p> <p>—代码断点</p>
watch	<p>监视某个表达式或变量，当它被读或被写时让程序中断。格式如下： watch EXPRESSION</p> <p>—数据断点</p>
set	<p>修改变量值。格式如下： set variable=value</p>
step	<p>单步执行，进入遇到的函数。</p>
next	<p>单步执行，不进入函数调用，即视函数调用为普通语句。</p>
continue	<p>恢复中断的程序执行。</p>
help	<p>通过下面的方法获得帮助，下例为获得list指令。 help list</p>

GNU调试器GDB简介 (5/5)

□ GDB中的断点有四种状态:

- ✓ 有效(Enabled)
- ✓ 禁止(Disabled)
- ✓ 一次有效(Enabled once)
- ✓ 有效后删除(Enabled for deletion)

□ 条件断点的设置语句:

```
(gdb) break ...if COND
```

GDB 反向调试 (Reverse Debugging)

□ GDB7.0提供了反向调试功能

指令	说明
reverse-continue	反向运行程序直到遇到使程序中断的事件，如断点、观察点、异常等
reverse-step	反向运行程序到上一次被执行的源代码行。
reverse-stepi	反向运行程序到上一条机器指令
reverse-next	反向运行到上一次被执行的源代码行，但是不进入函数。
reverse-nexti	反向运行到上一条机器指令，除非这条指令用来返回一个函数调用、整个函数将会被反向执行。
set exec-direction [forward reverse]	设置程序运行方向，可以用平常的命令step和continue等来执行反向的调试命令。

□ 反向调试的基本原理为 record/replay

□ 使用 record 命令进行录制，使用record stop关闭记录

课程大纲

 软件调试原理概述

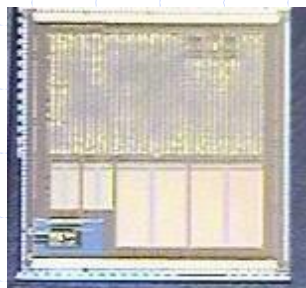
 嵌入式软件调试概述

 GDB Server分析

 KGDB简介

概述

- ❑ 嵌入式软件开发过程中常见的错误和通用PC机上软件开发中常见错误有着较大的区别。
- ❑ 常见错误分类如下
 - 硬件相关错误
 - 通讯相关错误
 - 多任务相关错误
 - 用户界面相关错误
 - 内存访问相关错误



硬件相关错误

- 嵌入式软件开发过程中，常出现硬件相关错误，例如：
 - WYNOT错误（“Worked Yesterday, NOt Today”）
 - 评估板上运行正确、现场运行错误
 - 低频运行正确、高频运行错误
 -
- 对于仅经历过纯软件编程培训，对数字电路知识甚少的软件工程师而言，首先需准确区分这些错误是由于软件设计产生的错误，还是由于硬件问题所导致。
- 对专用集成电路（ASIC）理解错误：
 - 寄存器设置、高速缓存（cache）控制、中断/内存控制
- 与信号干扰有关

通讯相关错误

- 由于嵌入式设备需要与PC、各种专用设备，以及其他嵌入式子系统进行通信，可能采用标准的协议：
 - USB
 - UART
 - PCI
 -
- 也可能采用各种专用的协议，出现与标准协议不兼容等异常情况。

多任务相关错误

- ❑ 由于多任务开发中的缺陷所产生的问题，是嵌入式系统开发中最难解决的问题之一，直接影响到系统的可靠性、健壮性、执行效率和可维护性。
- ❑ 嵌入式多任务环境下可能出现的问题包括：
 - 划分问题：任务～中断
 - 优先级设置问题：任务～任务、任务～中断
 - 同步问题：中断～任务、任务～任务
 - 互斥问题：中断～任务、任务～任务
 - 通讯问题：中断～任务、任务～任务
 - 异常处理问题：出错与恢复、执行任务取消



其他错误

□ 用户界面相关错误

- 由于输入、输出设备的多样性，所导致的错误
- 表现直观，是最好解决的一类错误。

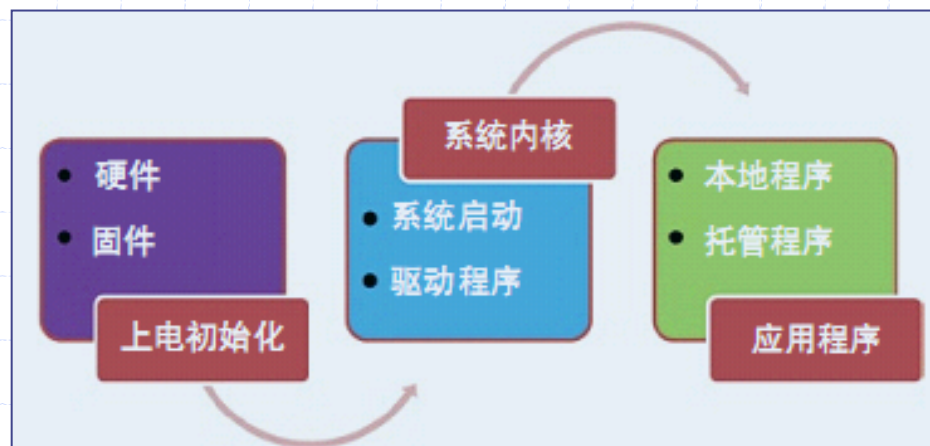
□ 内存访问相关错误

- 非法访问：非法修改内核、其他任务的内存空间
- 内存碎片问题：无法进行内存碎片整理，导致空闲内存总数够，但连续内存数目不够情况。



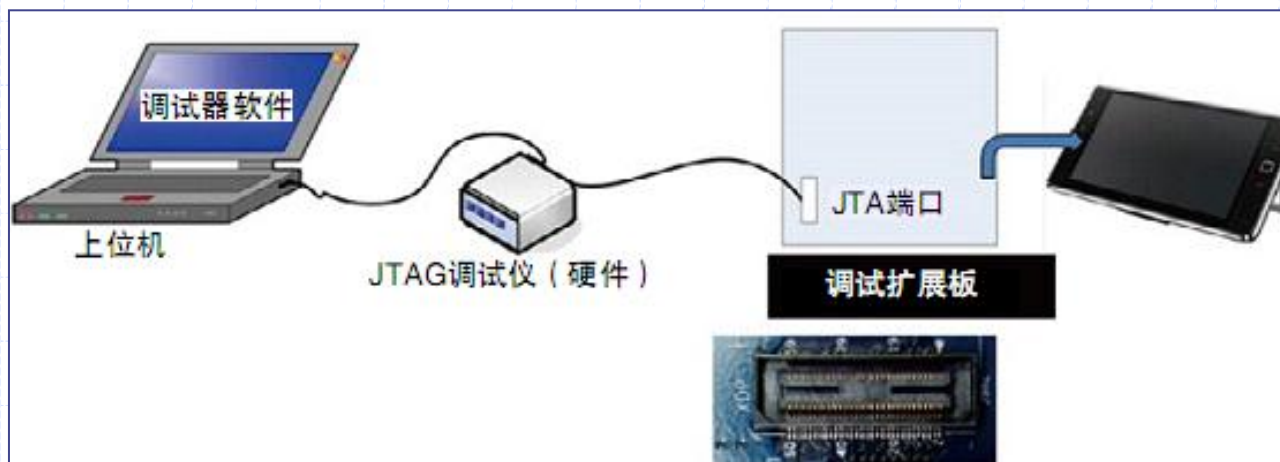
嵌入式系统调试的三个层次

- 从嵌入式系统开发和执行过程看，嵌入式系统调试可以分为三个层次
 - 上电（power-on）初始化调试：硬件、固件
 - 系统内核调试：bootloader、设备驱动程序、操作系统
 - 应用程序调试



上电初始化调试概述

- 系统上电和处理器复位后，首先执行的是固化在**ROM**或者**flash**上的硬件初始化程序，即通常所说的固件（**firmware**）。
- 在这一阶段，因为运行环境很简陋，软件调试器还无法工作，所以比较有效的一种调试手段是使用基于**JTAG**协议的硬件调试器（**JTAG**调试器）。



JTAG简介

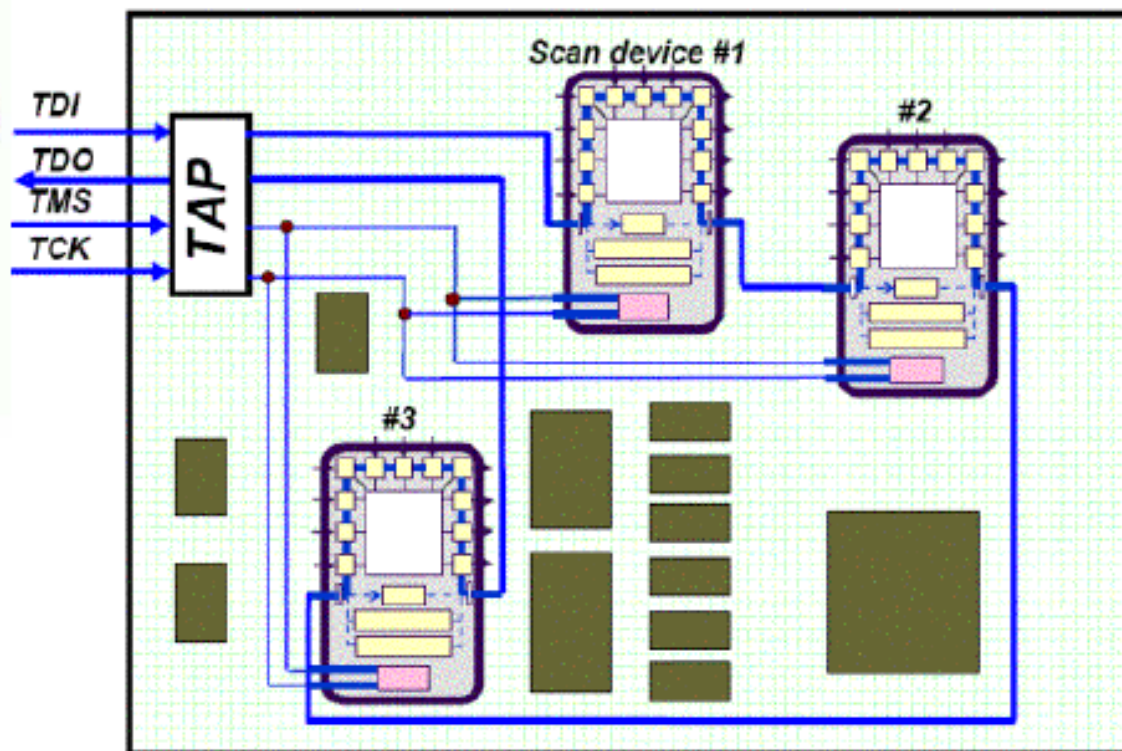
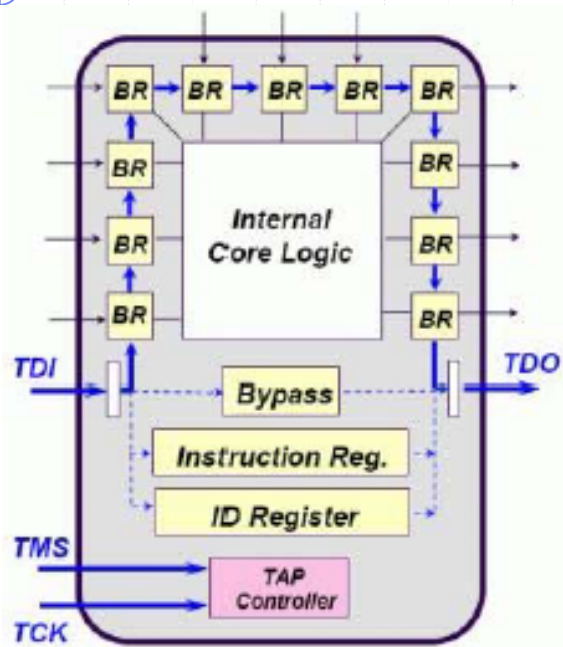
- JTAG(Joint Test Action Group, 联合测试行动小组)是1985年制定的检测PCB和IC芯片的一个标准。
- 1990年被修改后成为IEEE的一个标准, 即IEEE1149.1-1990。
- 通过这个标准, 可以对具有JTAG接口的芯片硬件电路进行边界扫描和故障检测。

JTAG基本原理

□边界扫描原理

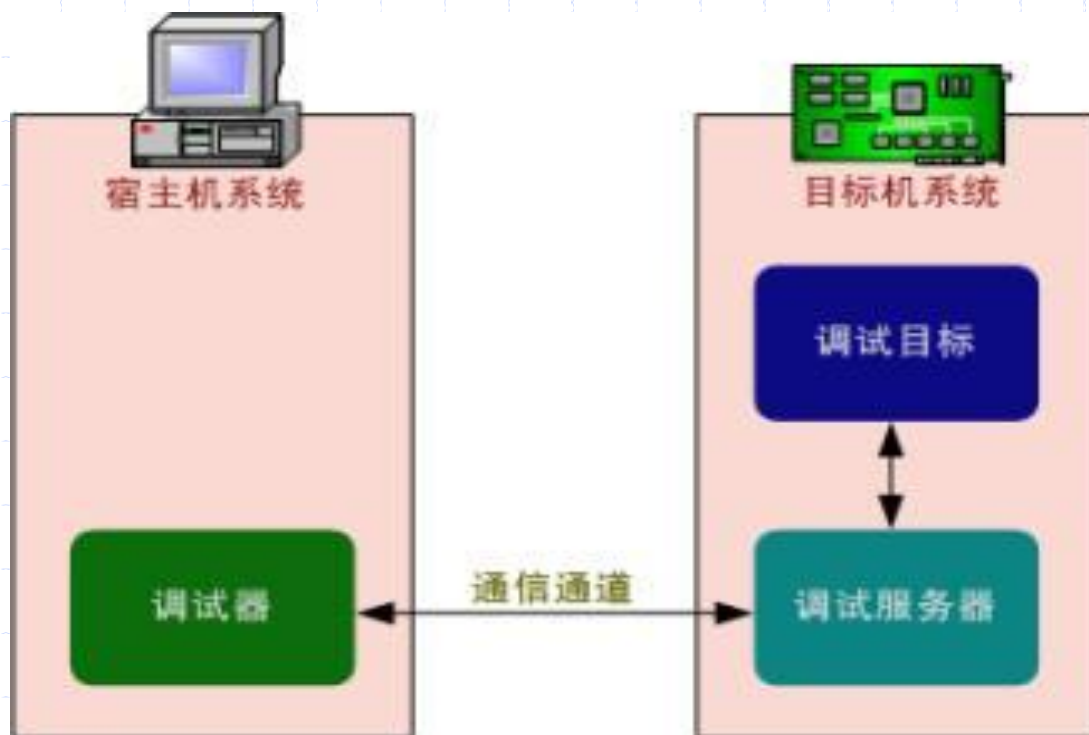
- 在芯片的输入输出管脚增加一个移位寄存器单元，称为边界扫描寄存器（BR），这些边界扫描寄存器可以将芯片和外围的输入输出隔离开来。
- 通过边界扫描寄存器单元，可以实现对芯片输入输出信号的观察和控制。
- 对于芯片的输入管脚，通过与之相连的边界扫描寄存器单元把信号（数据）加载到该管脚中去；
- 对于芯片的输出管脚，也可以通过与之相连的边界扫描寄存器“捕获”（CAPTURE）该管脚上的输出信号。

JTAG接口应用举例



嵌入式软件调试 vs 通用软件调试

- 通用软件调试与嵌入式软件调试环境上存在明显差别
- 嵌入式软件采用远程调试模式（或交叉调试模式），即：调试器运行于通用桌面操作系统，而被调试的程序（目标程序）则运行于特定硬件平台

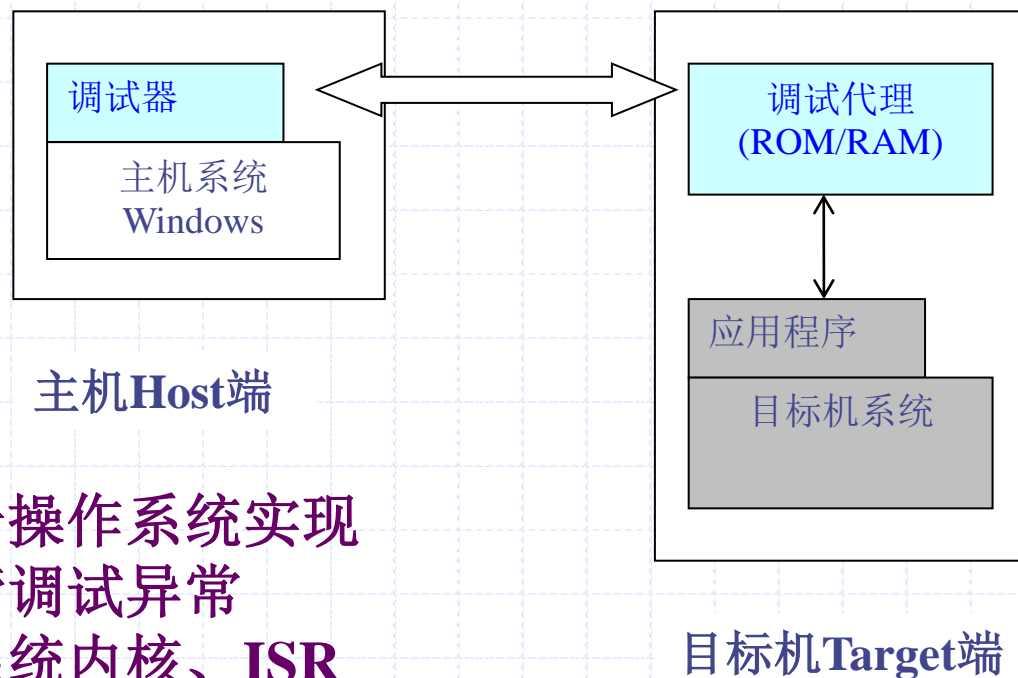


远程调试特点

- ✓ 调试器和被调试程序运行在不同机器上
- ✓ 目标机上具备某种形式的调试代理
- ✓ 符号表驻留在主机端，目标机端工作在无符号状况下
- ✓ 调试器通过某种端口（串口、网络、JTAG），按预定的通信协议与目标机端的调试代理建立联系
- ✓ 支持ROM型代码断点调试

远程调试模式—系统级调试

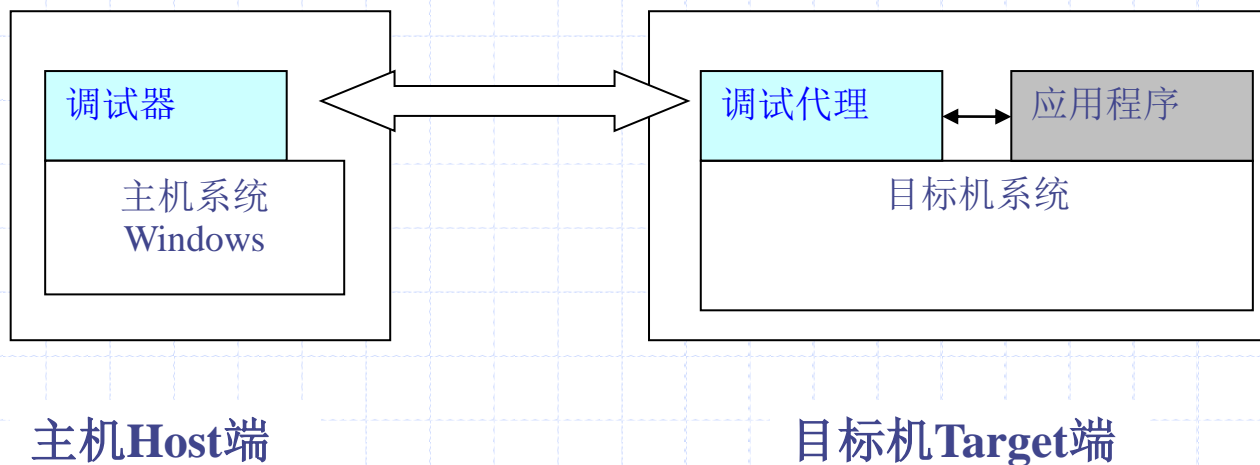
根据调试器对被调试程序的调试能力分类，可以把远程调试器模型分为：系统级调试和任务级调试，下面是系统级调试原理示意图：



特点：

- ✓ 调试器不依赖于操作系统实现
- ✓ 调试器直接接管调试异常
- ✓ 可以调试操作系统内核、ISR
- ✓ 但无法调试单个任务

远程调试模式—任务级调试



特点:

- ✓ 调试器基于操作系统功能实现
- ✓ 操作系统接管调试异常，再转交调试器
- ✓ 可以调试多任务情况
- ✓ 无法调试操作系统内核、ISR

GDB远程调试解决方案

- GDB为远程调试提供了两种解决方案：GDB Server、GDB Stub
 - ✓ GDB Server对Unix/Linux操作系统具有依赖性，只能完成应用程序级调试—任务级调试；
 - ✓ GDB Stub不依赖于操作系统，可对整个系统进行调试—系统级调试。

GDB远程调试通讯协议

- ❑ GDB支持远程串口通信协议RSP（Remote Serial Protocol），及网络通信协议。
- ❑ RSP是一种字符流协议，所有的调试命令和调试应答信息以可阅读的字符串方式传输。
- ❑ RSP由‘\$’、调试指令/数据、‘#’符号，以及校验码组成，即“\$packet-data#checksum”。校验码为2位无符号十六进制数，以调试指令/数据字符之和取256的模。
- ❑ 主机或目标机在接收到GDB报文时，首先对报文进行校验，判断报文是否正确，如果正确，发送‘+’到发送方确认；如果检查出错，通过向发送方发送‘-’要求重新发送。

GDB远程调试指令

- ❑ GDB拥有一批远程调试指令，主要包括： 'c', 's', 'm', 'M', 'g', 'G'等六个基本命令：
 - ✓ 'm addr,length': 读取内存值
 - ✓ 'M addr,length:XX...': 写内存值
 - ✓ 'g': 读取寄存器值
 - ✓ 'G XX...': 写寄存器值
 - ✓ 'c addr': 继续执行程序
 - ✓ 's addr': 单步执行程序
- ❑ 通过基本命令组合，可以实现调试器的基本功能，如：设置断点采用m、M。

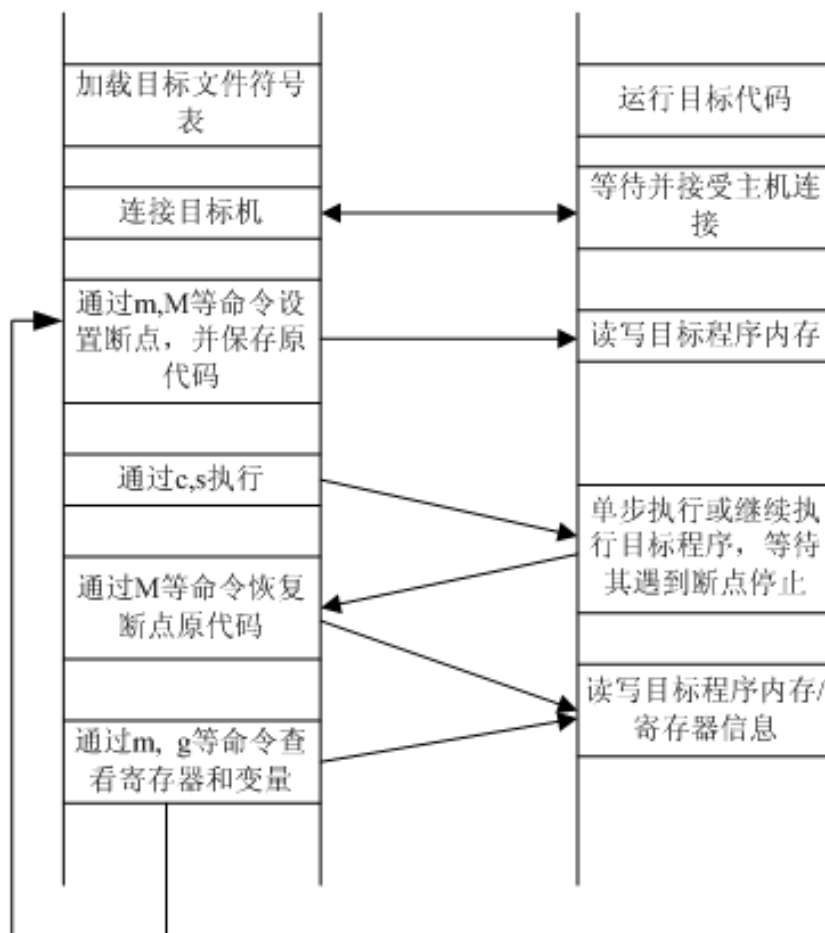
GDB远程调试流程

GDB

主机端

目标机端

GDB Server/
GDB Stub



课程大纲

-  软件调试原理概述
-  嵌入式软件调试概述
-  GDB Server分析
-  KGDB简介

GDB Server能做什么？

- ❑ GDB Server能够远程调试类Unix系统下的应用程序。
- ❑ GDB Server通过类Unix系统提供的ptrace系统调用来实现对被调试应用程序的访问和控制。
- ❑ GDB Server对操作系统具有依赖性，导致它无法实现操作系统级调试，只能实现应用程序级调试。
- ❑ GDB Server的优势在于：应用程序代码无需与调试模块链接就能实现远程调试。

GDB Server主要工作流程

- ❑ GDB Server通过通讯端口与GDB建立连接。
- ❑ GDB Server与被调试程序绑定/关联：让被调试程序成为GDB Server的子进程
 - ✓ 被调试程序未运行：创建被调试程序子进程，调用ptrace的PTTRACE_TRACEME声明GDB Server为它的跟踪父进程。
 - ✓ 被调试程序已运行：GDB Server通过调用ptrace的PTTRACE_ATTACH向系统声明跟踪被调试程序。
- ❑ GDB通过GDB Server设置断点等信息。
- ❑ 被调试程序运行产生异常，操作系统通知GDB Server接管。
- ❑ GDB通过读写被跟踪程序的指令空间、数据空间或寄存器，完成用户调试命令。

如何绑定未运行的被调试程序？（1/3）

```
int
create_inferior (char *program, char **allargs)
{
    int pid;

    pid = fork ();
    if (pid < 0)
        perror_with_name ("fork");

    if (pid == 0)
    {
        int pgrp;

        /* Switch child to it's own process group so that signals won't
           directly affect gdbserver. */

        pgrp = getpid ();
        setpgroup (0, pgrp);
        ioctl (0, TIOCSGRP, &pgrp);

        ptrace (PTRACE_TRACEME, 0, (PTRACE_ARG3_TYPE) 0, 0);

        execv (program, allargs);

        fprintf (stderr, "GDBserver (process %d): Cannot exec %s: %s.\n",
                 getpid (), program,
                 errno < sys_nerr ? sys_errlist[errno] : "unknown error");
        fflush (stderr);
        _exit (0177);
    } ? end if pid==0 ?

    return pid;
} ? end create_inferior ?
```

GDB Server
创建子进程

子进程设置为
被跟踪模式
创建被跟踪子
程序

如何绑定未运行的被调试程序？（2/3）

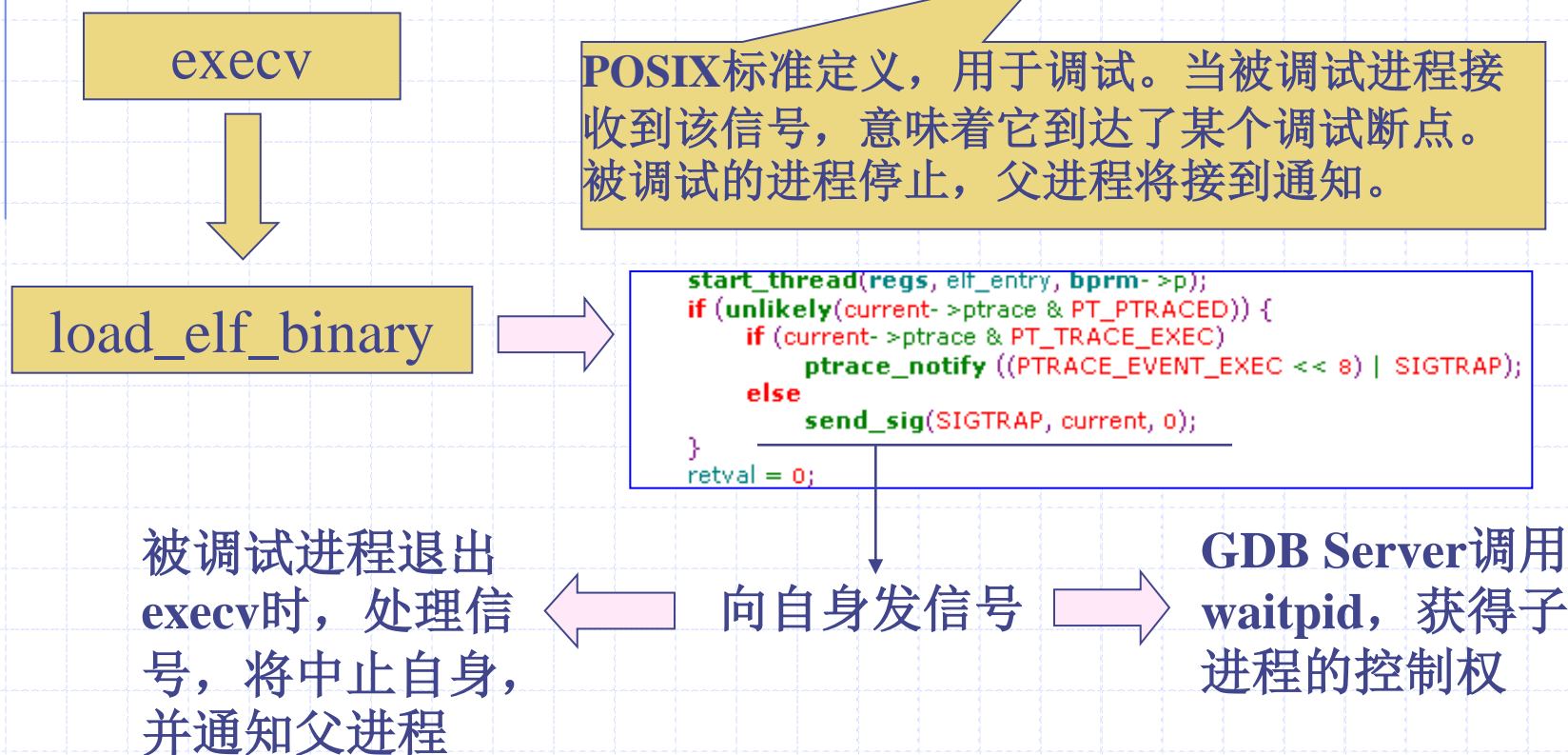
```
asmlinkage int sys_ptrace(long request, long pid, long addr, long data)
{
    struct task_struct *child;
    struct user *dummy = NULL;
    int i, ret;
    unsigned long __user *datap = (unsigned long __user *)data;

    lock_kernel();
    ret = -EPERM;
    if (request == PTRACE_TRACEME) {
        /* are we already being traced? */
        if (current->ptrace & PT_PTRACED)
            goto ↓out;
        ret = security_ptrace(current->parent, current);
        if (ret)
            goto ↓out;
        /* set the ptrace bit in the process flags. */
        current->ptrace |= PT_PTRACED;
        ret = 0;
        goto ↓out;
    }
    ret = -ESRCH;
    read_lock(&tasklist_lock);
    child = find_task_by_pid(pid);
    if (child)
        get_task_struct(child);
    read_unlock(&tasklist_lock);
    if (!child)
        goto ↓out;|
```

→ 设置任务控制块中的被跟踪标记

如何绑定未运行的被调试程序？（3/3）

- 调用 `execv` 系统调用的进程，发现已经被跟踪，则把代码装入系统后，向自身发送信号 **SIGTRAP**。



如何绑定已运行的被调试程序？（1/2）

```
void
linux_attach_lwp (int pid, int tid)
{
    struct process_info *new_process;

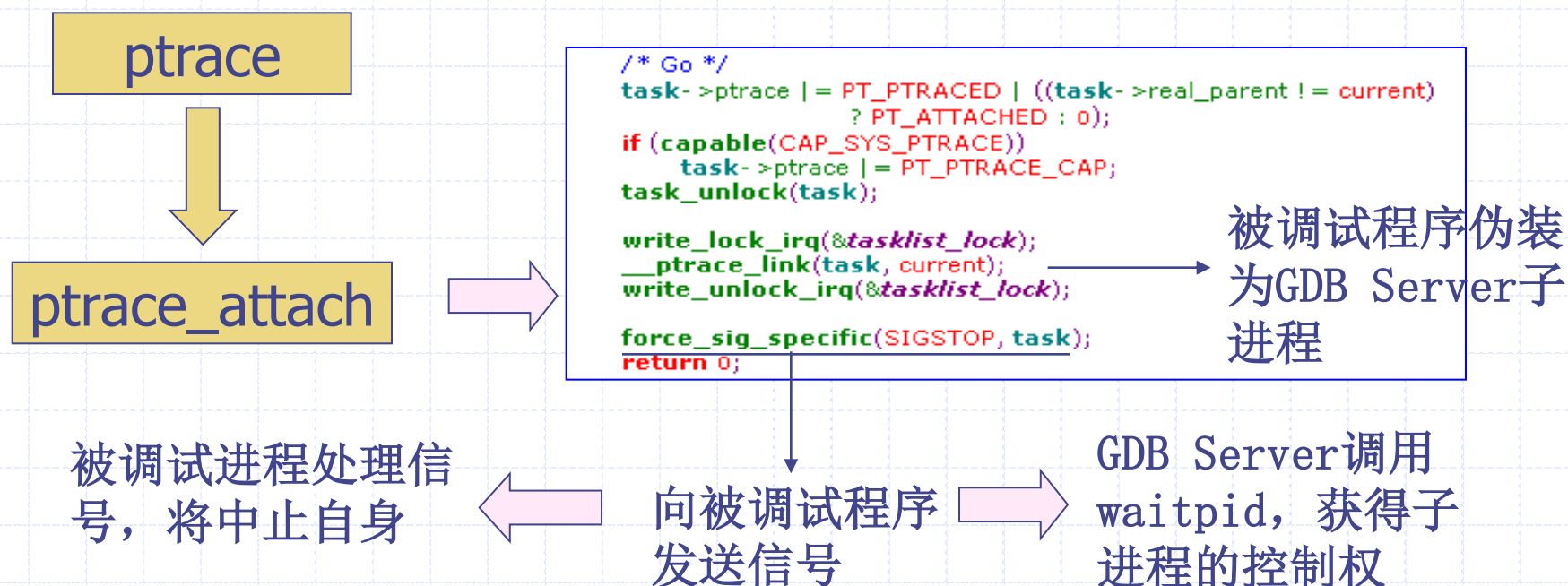
    if (ptrace (PT_TRACE_ATTACH, pid, 0, 0) != 0)
    {
        fprintf (stderr, "Cannot attach to process %d: %s (%d)\n", pid,
                 errno < sys_nerr ? sys_errlist[errno] : "unknown error",
                 errno);
        fflush (stderr);

        /* If we fail to attach to an LWP, just return. */
        if (!using_threads)
            _exit (0177);
        return;
    }
}
```

已运行进程设置
为被关联模式

如何绑定已运行的被调试程序？（2/2）

- `ptrace`系统调用判断关联请求合法后，将需调试的进程伪装为GDB Server的子进程，并向它发送SIGSTOP信号中止运行，使父进程得到控制权。



课程大纲

-  软件调试原理概述
-  嵌入式软件调试概述
-  GDB Server分析
-  KGDB简介

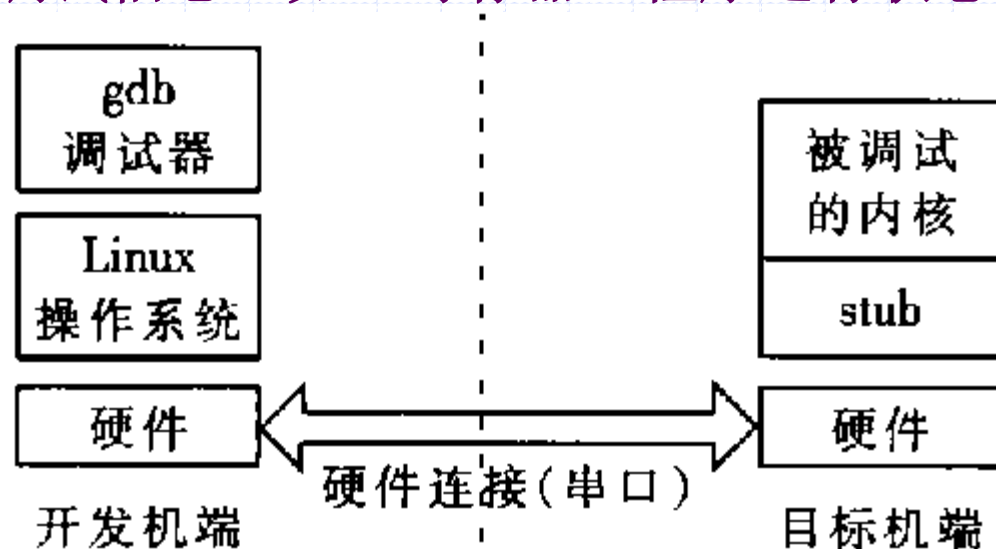
关于KGDB

□ KGDB在Linux内核上提供调试器功能补丁，属于GDB Stub类型，是一种内核调试器，实现系统级软件调试，适用于下述情况：

- ✓ 开发者需要调试操作系统内核
- ✓ 开发者需要调试设备驱动模块

GDB Stub（插桩）技术简介

- ❑ KGDB采用了插桩（stub）技术实现系统级调试。在目标操作系统内核中加入调试功能模块（插桩），如：通信模块、异常处理模块等。
- ❑ 插桩的基本思想是：接管操作系统的中断或者异常处理，使得当中断或者异常发生时，调试器获得整个系统的控制权，进而获取当前的调试信息，如：寄存器、程序运行状态等信息。



KGDB功能分析

□ KGDB主要有二大模块组成：

- ✓ 初始化模块：完成初始化过程，接管系统的所有异常、设置串口通信等低层实现；
- ✓ 控制模块：实现通信，对GDB发送的信息包进行解析并执行，对应答包进行打包发送。

KGDB初始化模块实现

□ 接管系统异常处理

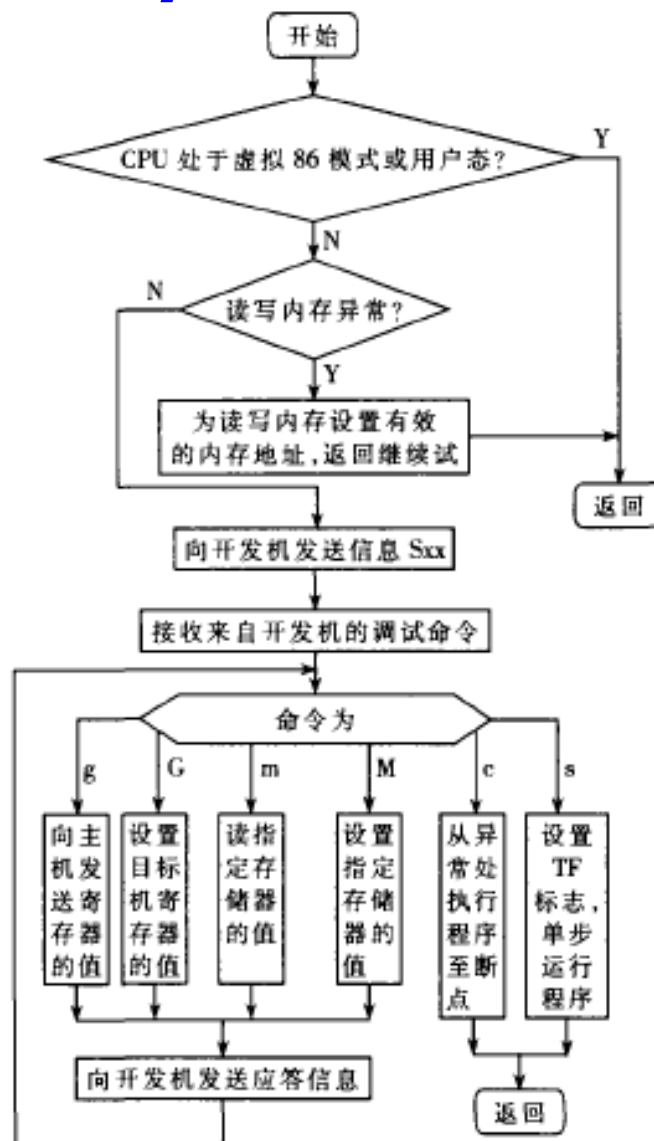
- ✓ KGDB对各个需要捕获的异常处理函数进行修改。当发生异常时，使异常处理事件进入统一的异常处理函数`handle_exception`。
- ✓ `void set_debug_traps(void)`向系统注册调试过程中的处理函数：`handle_exception ()`。

□ 进行串口初始化

- ✓ `struct serial_state* gdb_serial_setup(int ttyS, int baud)` 参数`ttyS`为串口号，`baud`为传输波特率。函数返回该串口的状态。

KGDB控制模块实现 (1/2)

异常处理函数: handle_exception



KGDB控制模块实现（2/2）

□ 相关通讯处理函数：

- ✓ `static int read_char(void)` 从串口读取一个字节的数据。
- ✓ `static void write_char(int chr)` 向串口写一个字节的数据。
- ✓ `static void getpacket(char *buffer)` 根据串口通讯协议RSP，获取一个命令帧数据，并进行分析。
- ✓ `static void putpacket(char *buffer)` 根据串口通讯协议RSP，组装一个命令帧数据，并发送。

注：上述串口通讯函数只能以轮询方式收/发数据。

内核进入调试状态的路径

□ 内核进入调试状态有两种方法：

- ✓ 方法一：通过在内核启动的时候向内核传入参数，可以调试系统启动过程内核的运行状况；参数（Gdb、gdbttyS、gdbbaud）
- ✓ 方法二：在内核完全导入系统正常运行的情况下，通过使用一个gdbstart工具将驱动串口设备，内核的控制权交给本地主机。



谢谢!

