

嵌入式系统

An Introduction to Embedded System

第二课 嵌入式系统开发概述

课程安排的三个思路

- 了解嵌入式处理器总体架构及特点
- 掌握嵌入式实时操作系统概念及相关理论
- 熟悉基于Linux的嵌入式软件系统开发技术

课程大纲

 嵌入式处理器的补充介绍

 嵌入式系统开发方法概述

 嵌入式系统软件开发概述

 Linux操作系统、虚拟机概述

 GNU开发工具链简介

 嵌入式软件开发环境建立实验

嵌入式处理器几大类体系结构

- ARM
- MIPS
- POWER
PC
- X86

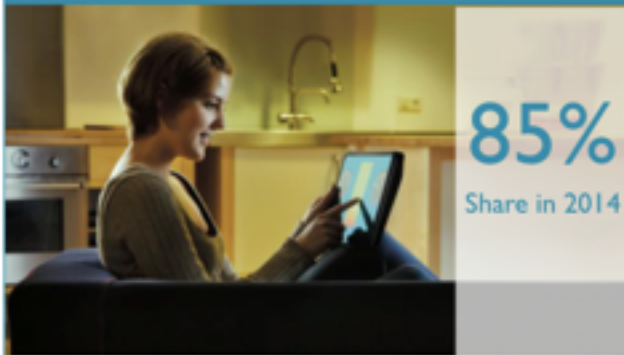
ARM体系结构

- **Advanced RISC Machine**
 - 高性能、廉价、耗能低的RISC处理器，是工业标准
 - 只设计内核的英国公司
 - 1985年英国剑桥ARM原型，1990年成立ARM公司
 - 不制造VLSI设备，只提供授权
 - 2010年，市场占有率75%
 - ARM内核被授权给数百家厂商
- ARM主要应用于无线局域网、3G、手机、手持设备、有线网络通讯
- 应用形式：集成到专用芯片中作控制器
- 集成ARM内核的芯片
- Intel、TI、ADI、三星、Motolora、飞利浦半导体、安捷伦、高通、Atmel、Intersil、Alcatel、Altera、Cirrus Logic、Linkup、Parthus、LSI Logic、Micronas、Silicon Wave、Virata、Portalplayer inc.、NetSilicon、Parthus

ARM Q4 2014

ARM's Main Growth Markets

Mobile Computing



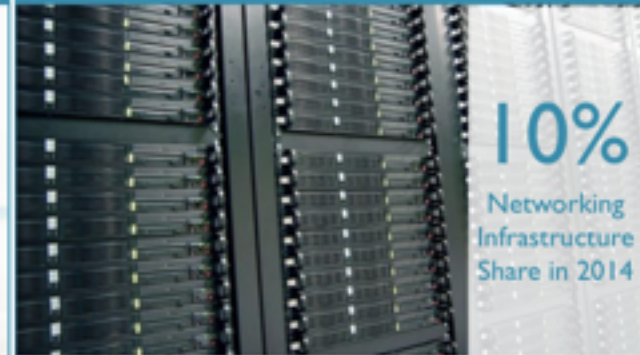
- Smartphones, tablets and laptops
- Apps processor, modem, connectivity, touchscreen and image sensors
- Apps processor: Increasing proportion using ARM technology with higher royalty per chip from ARMv8-A, octa-cores, graphics and physical IP

Embedded Intelligence



- White-goods, wearables, smart devices in industrial, transport and utilities
- Microcontrollers, smartcards, embedded connectivity chips
- 200 companies have licenced ARM processors for use in embedded intelligent devices

Enterprise Infrastructure



- Base stations, routers, switches, and servers for cloud and data centres
- Networks evolve to cope with increased data at lower latency: virtualisation, integration and programmability
- Most major chip vendors have announced ARM-based products

MIPS体系结构

- **Microprocessor without Interlocked Piped Stages:** 无互锁流水级的微处理器
- 是高性能、高档次的RISC架构的嵌入式微处理器
- MIPS体系结构起源于Stanford大学的研究
- 1984年创立MIPS公司，不生产芯片，只卖授权
- 应用于消费电子、网络、宽带、智能卡、数字电视



PPC体系结构

- PowerPC
 - Motorola半导体（Freescale）联合IBM、苹果电脑设计
 - 是高性能、高档次的RISC架构的嵌入式微处理器
 - 产品：MPC680、8245、8260
- 应用于DSL调制解调器、SOHO路由器、远程接入服务器、DSLAM、执行局交换机设备、无线基站、企业路由器、航空电子



X86体系结构

- Intel X86体系结构
- AMD的X86体系结构嵌入式处理器产品为Geode系列处理器
- CISC指令集



DSP

- DSP应用
 - 数字信号处理
 - 限时完成
- DSP处理器对系统结构和指令进行了特殊设计，使其适合于执行DSP算法，编译效率较高，指令执行速度也较高。
- 在数字滤波、FFT、谱分析等方面DSP算法正在大量进入嵌入式领域，DSP应用正从在通用单片机中以普通指令实现DSP功能，过渡到采用嵌入式DSP处理器。
- 代表性的产品
 - TI(Texas Instruments)的TMS C3X/C4X/C6X系列
 - ADI(Analog Device Inc.)的21X系列、Black Fin系列
- 很流行ARM+DSP结构

常用的多核结构

- 同构多核：ARM+ARM
 - 两个核功能不一样
 - 汽车：一是GPS全球卫星定位，二是人性化便捷式操作界面
- 异构多核：ARM+DSP
 - ARM管软件，DSP管数据处理（协处理器）
- 智能手机：ARM跑wince，DSP跑2.5G或3G协议栈
- ARM+FPGA
 - ARM管软件，FPGA重构为特殊功能的硬件

处理器评价指标

分类	因素
芯片能力	处理器性能 逻辑性能 存储器性能 支持高速算术 支持I/O和通信 安全特性和支持安全引导 处理器和逻辑部分之间的带宽 处理器和逻辑部分之间的延迟
商业因素	物料表 开发成本（见下面） 集成（和隐形成本） 芯片供应的长期性和技术支持 质量和可靠性 实现现场升级的容易程度（和成本）
设计与开发	进入市场的时间 快速、便捷和可靠的设计流 集成的验证 支持团队设计流 支持设计重用 支持工业标准的设计格式 支持所需的设计入口方法 文档和技术支持
芯片物理特征	物理尺寸 功耗 易于集成和产生的PCB的复杂程度 可连接性 耐久性 是否对辐射敏感 支持的温度范围
灵活性	可伸缩性（有更大或更小的芯片，而且只需要很小的甚至不需要工作就可以调整） 可移植性（用标准格式设计可以移植到其他平台上或从其他平台移植过来） 再编程能力（可以现场甚至在运行时刻动态改变功能） 易于划分（能在硬件和软件之间划分功能） 可扩展性（易于集成新的功能）

嵌入式处理器性能评价

- 度量项目

- 性能指标

- 吞吐率

- 实时性

- 各种资源利用率

- 可靠性

- 平均故障间隔时间MTBF

- 可维护性

□ 评价方法

1、测量法

- 采样

- 跟踪

2、模型法

- 分析模型法

- 模拟模型法

嵌入式处理器评估主要指标

- MIPS测试基准（millions of instructions per second）
- Whetstone测试基准
 - 测试浮点运算性能，适用于科学计算、工程应用系统等性能评估
- Dhrystoen测试基准
 - 整数和逻辑运算性能测试
- EEMBC测试向量（嵌入式微处理器基准测试协会）
- 非赢利性组织
- 公布了**46**个性能测试向量，用于电信、网络、消费电子、办公设备、汽车电子等领域
- 考虑了编译器对代码大小、执行效率的影响
- 其他测试
 - BDTI基准测试程序：DSP性能评估
 - MediaBench基准测试程序：多媒体处理性能评估

课程大纲

 嵌入式处理器的补充介绍

 嵌入式系统开发方法概述

 嵌入式系统软件开发概述

 Linux操作系统、虚拟机概述

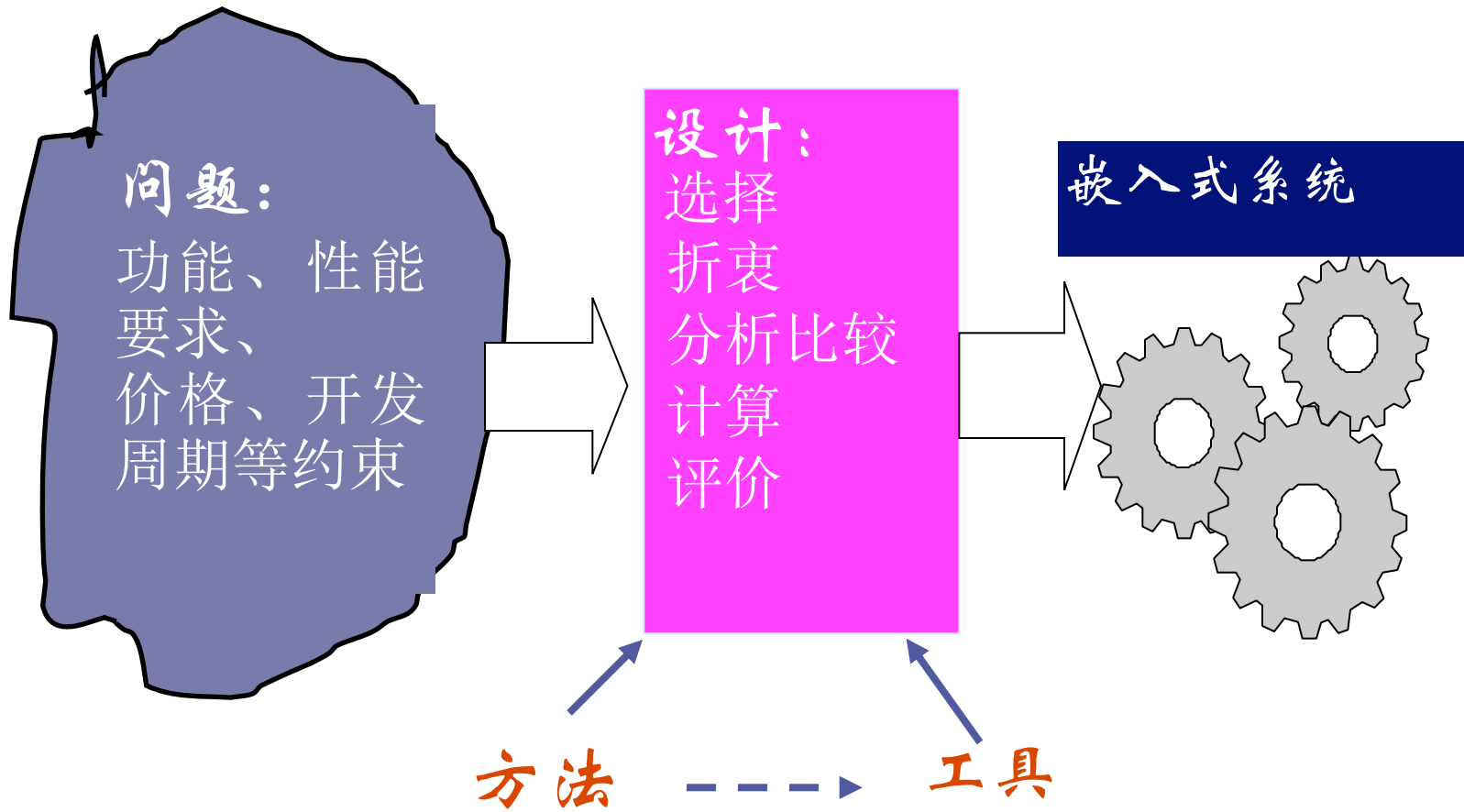
 GNU开发工具链简介

 嵌入式软件开发环境建立实验

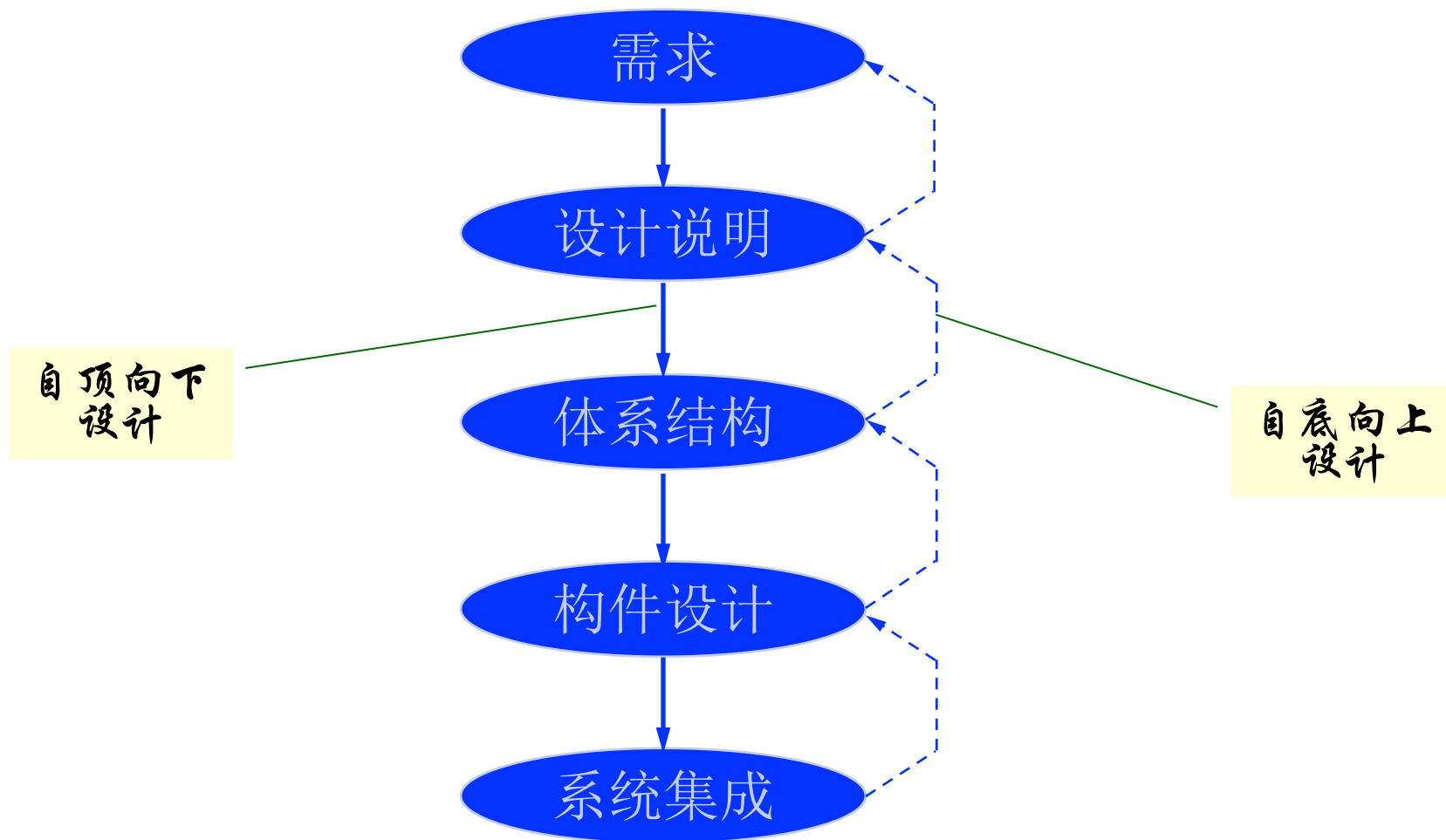
够好的软件，还要够快

- 如何能做出够好的软件，而不至于破产？
 - 需要能有效地开发（及测试）软件
- 遵循好的计划
 - 从客户的需求开始
 - 设计体系结构来定义系统的构件（任务、模块等）
 - 加上遗漏的需求
 - 失效检测、管理和日志
 - 实时性问题
 - 与固件标准手册的兼容性
 - 故障安全保障
 - 做出详细设计
 - 遵循良好的开发过程来实现代码
 - 经常做设计和代码的评审
 - 经常做测试（单元和系统测试，最好是自动进行的）
 - 用版本控制来管理变更
 - 事后总结来改进开发过程

嵌入式系统的设计



设计流程—自顶向下或自底向上设计



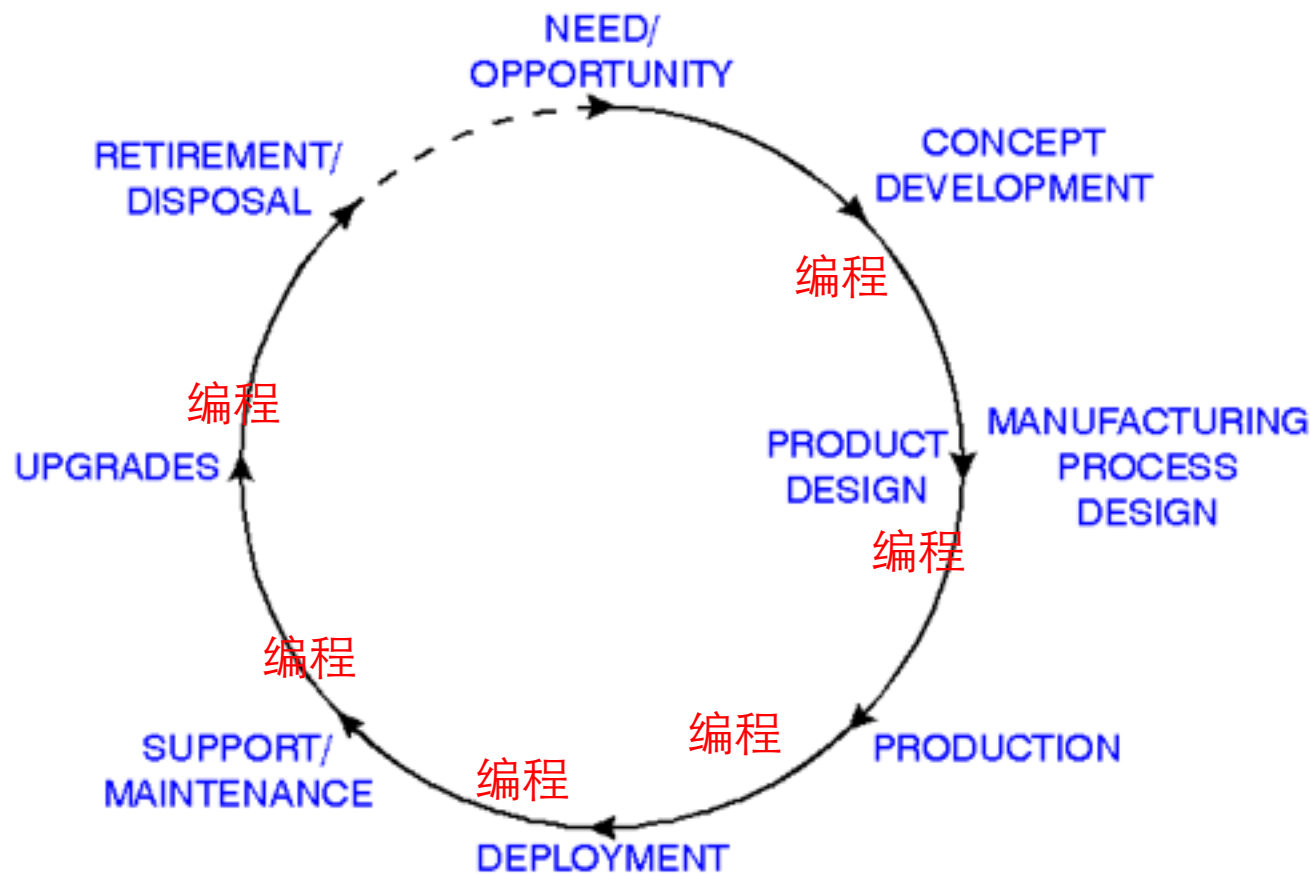
当计划遇上现实会怎样？

- 我们希望有一个强壮的计划，考虑到了可能的风险
 - 如果代码比预期的复杂很多怎么办？
 - 如果代码（或库）里有错误怎么办？
 - 如果系统没有足够的内存或吞吐能力怎么办？
 - 如果系统太贵了怎么办？
 - 如果主要开发人员离职了怎么办？
 - 如果主要开发人员不胜任、懒惰，或又笨又懒（还不肯离职）怎么办？
 - 如果团队中其他人生病了怎么办？
 - 如果客户增加了新的要求怎么办？
 - 如果客户想要提前两个月交货怎么办？
- 成功的软件工程有赖于很多因素的平衡取舍，很多因素都不是技术的！

软件生命周期概念

- 编程是软件开发最显而易见的阶段，但不是唯一的
- 在可以编程之前，必须知道
 - 代码必须做什么？需求定义
 - 代码要如何组织？设计定义
 - （只有在这之后才开始写代码）
- 如何能知道代码是可用的？测试计划
 - 在定义需求的时候就做计划是最好的
- 软件可能会随着时间的推移而变化——大量的顺势的修改的维护工作！
 - 纠错、适应、增强和预防性维护

产品开发生命周期



绘图：Phil
Koopman,
Carnegie Mellon
University

- 在整个代码开发和修改的过程中，值得多加努力来简化代码开发工作：理解、维护、改进和测试

需求

- **Ganssle关于嵌入式项目失败的第五个理由：模糊的需求**
- **需求的种类**
 - 功能性的——系统需要做的
 - 非功能性的——重要的系统属性，如响应时间、可靠性、能效、安全等。
 - 约束——对设计选择的限制
- **表现形式**
 - 文字——容易出现不完整、臃肿、含混不清，甚至矛盾的情况
 - 图表（状态图、流程图、消息序列图）
 - 精确
 - 常用作设计文档
- **可追溯性**
 - 每个需求应该能通过测试来验证
- **稳定性**
 - 需求变动会导致低效，而且常常会使问题产生“近因”效应（最近变更的需求变得最重要了）

例：GPS移动地图的需求

名称	GPS 地图
目的	为开车者提供用户级移动地图
输入	一个电源按钮，两个控制按钮
输出	背光 LCD，显示分辨率 400×600
功能	使用 5 种接收器的 GPS 系统；三种用户可选的分辨率，总是显示当前的经纬度
性能	0.25 秒内即可更新一次屏幕
生产成本	800 元
功耗	100mW
物理尺寸和重量	不大于 10cm×20cm，<350g



设计规格形式化描述

- ◆ 需求的精确描述
 - 描述明确，可理解
 - UML (Unified Modeling Language)
- ◆ 统一建模语言是可视化的设计说明语言
- ◆ 统一描述系统的硬件和软件
 - UML 可对系统的功能建模
 - 可自动产生实际设计的HDL 或C++ 代码

UML在嵌入式系统设计中的应用

- I-Logix公司的Rhapsody系列产品
- 基于统一建模语言UML的可视化编程环境
- 把UML各类视图映射为具体目标机程序语言
- 用于复杂实时嵌入式应用软件从分析、设计一直到代码实现和软件测试的开发过程
- NASA的火星探路者航天器就是运用Rhapsody在VxWorks上开发应用程序。

先设计再编程



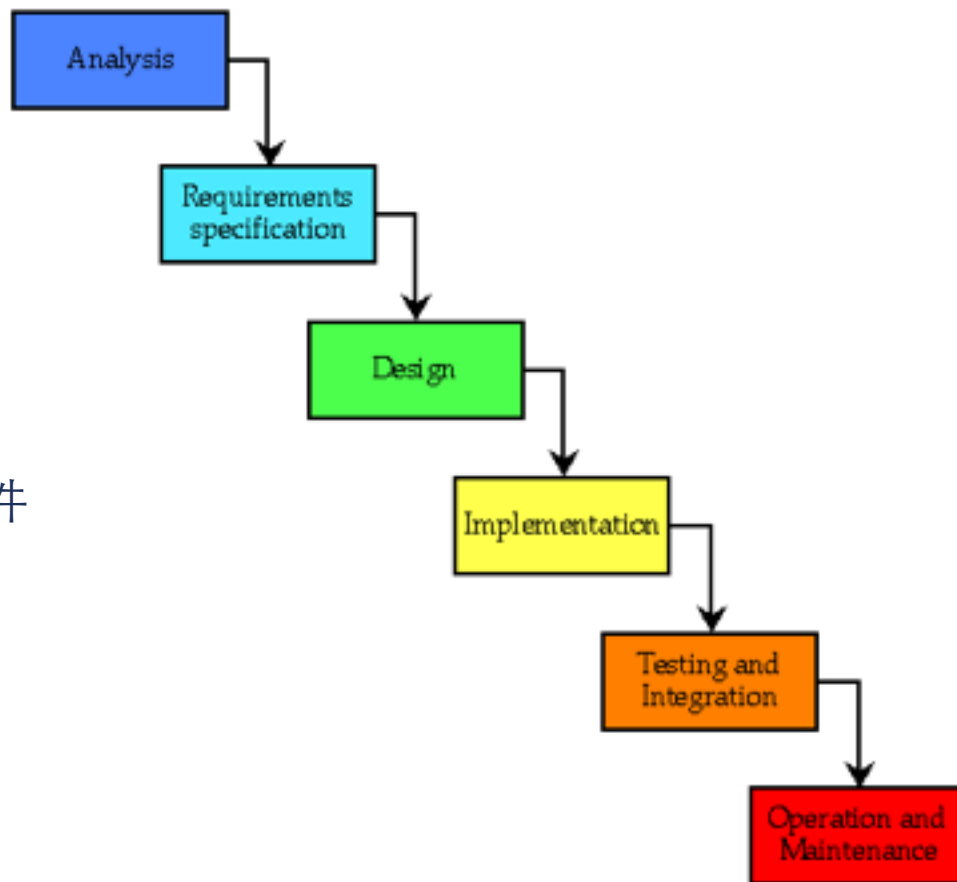
- **Ganssle**的第九个理由：过早开始编程
- 低估了所需软件的复杂度是非常常见的风险
- 写代码使你囿于特定的实现中
 - 太早开始会让你向隅而泣
- 先设计再编程的好处
 - 预先看到系统的复杂成都，对投入能做出更精准的估计和计划
 - 能用设计图而不是代码来讨论系统应该做什么和怎么做。**Ganssle**的第七个理由：不科学
 - 可以在文档中使用设计图来简化代码的维护，降低员工换岗的风险

开发模式

- 如何规划这些工作的时间？
- 考虑开发风险的量
 - 新的单片机？
 - 例外的需求（吞吐量、功率、安全认证等等）
 - 新产品？
 - 新客户？
 - 需求变更？
- 根据风险选择模式
 - 低：可以做详细规划，做完整的事先设计，瀑布式
 - 高：用交互或敏捷的开发模式，螺旋式，先做高风险部分的原型

瀑布（理想化的）

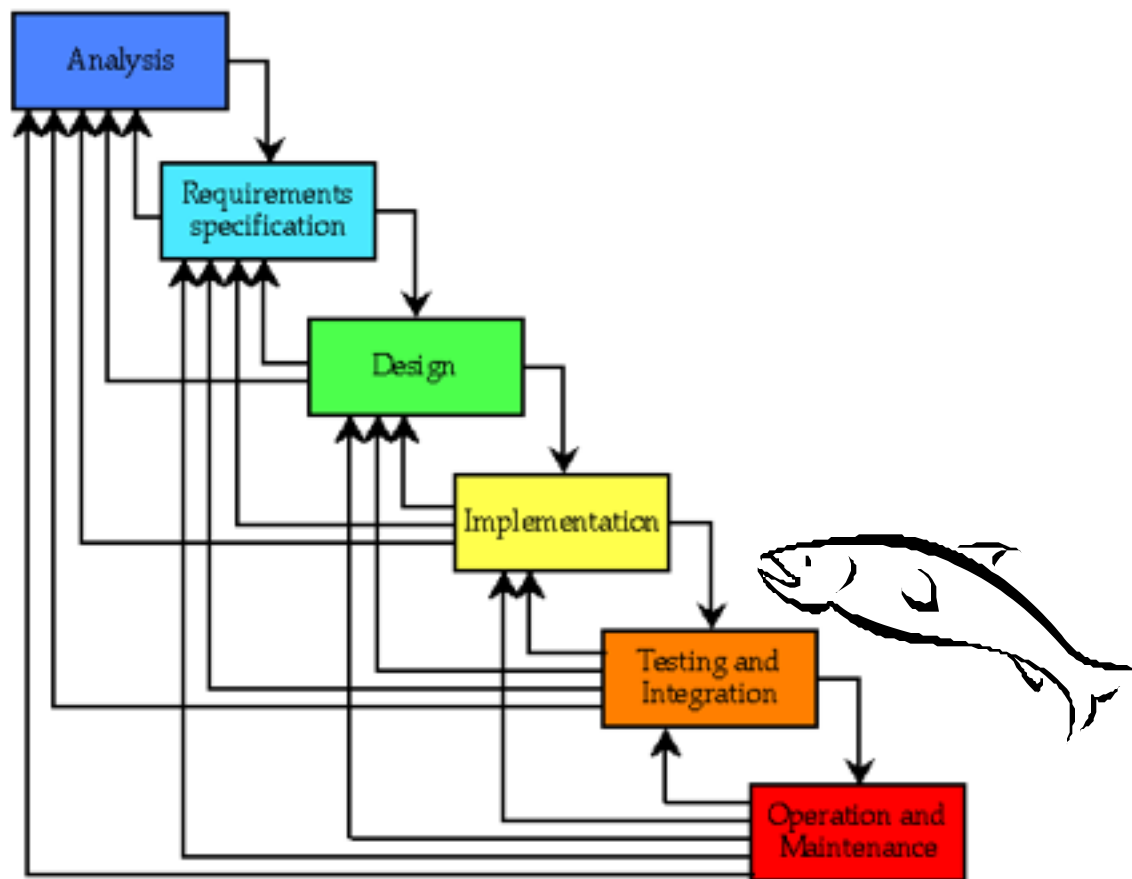
- 规划好工作，然后照规划来工作
- **BUFD: Big Up-Front Design**
事先规划好的设计
- 这个模式意味着我们和客户都
 - 预先知道所有的需求
 - 知道部件之间所有的交互
 - 知道要花多久来写软件和调试软件



绘图: Jon McCormack, Monash University

瀑布（实际的实现）

- 现实：我们并非无所不知，所以存在大量的返工



绘图: Jon McCormack, Monash University

敏捷开发：SCRUM

- <http://www.mountaingoatsoftware.com/topics/scrum>
- <http://www.codeproject.com/KB/architecture/scrum.aspx>

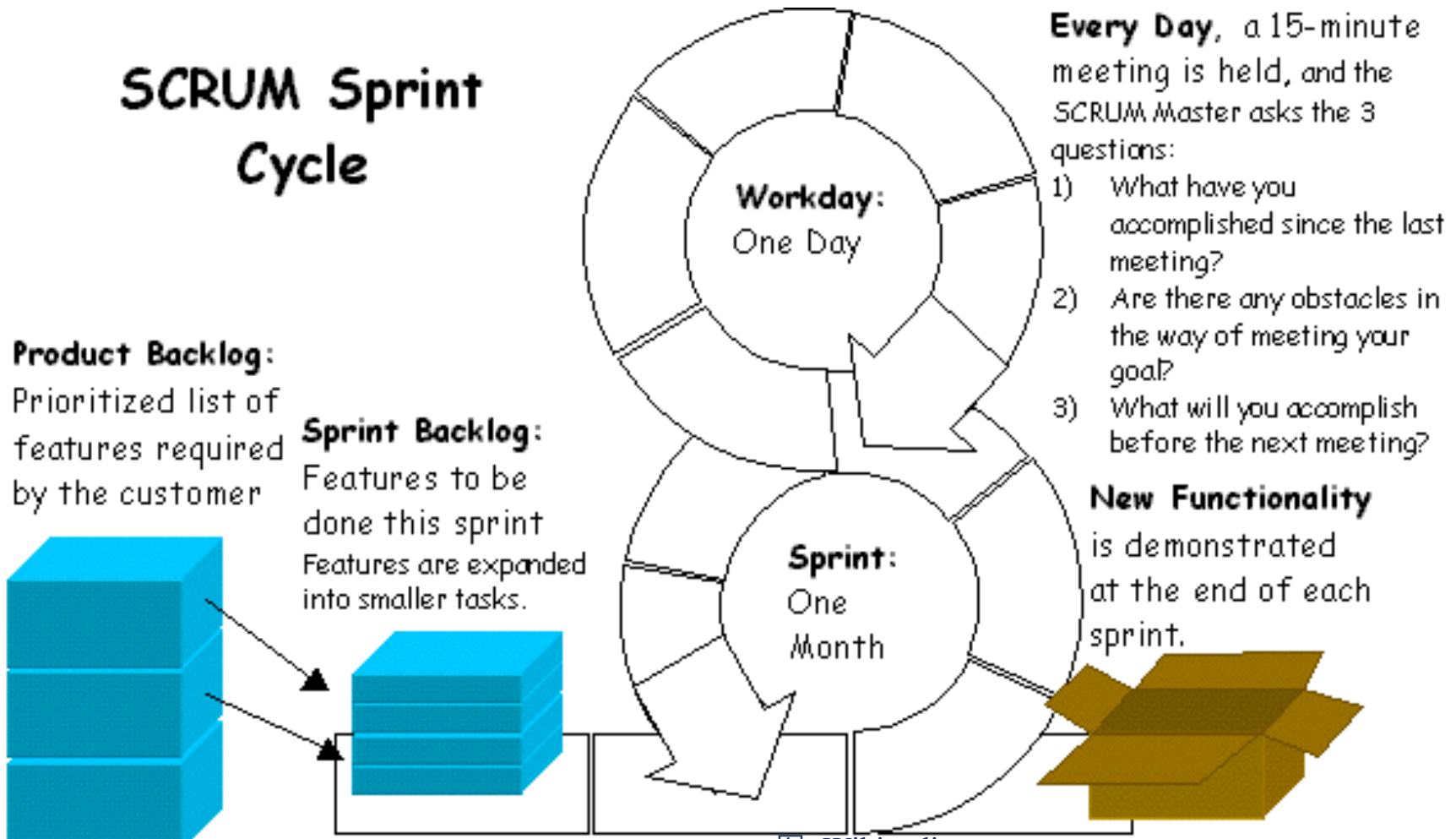
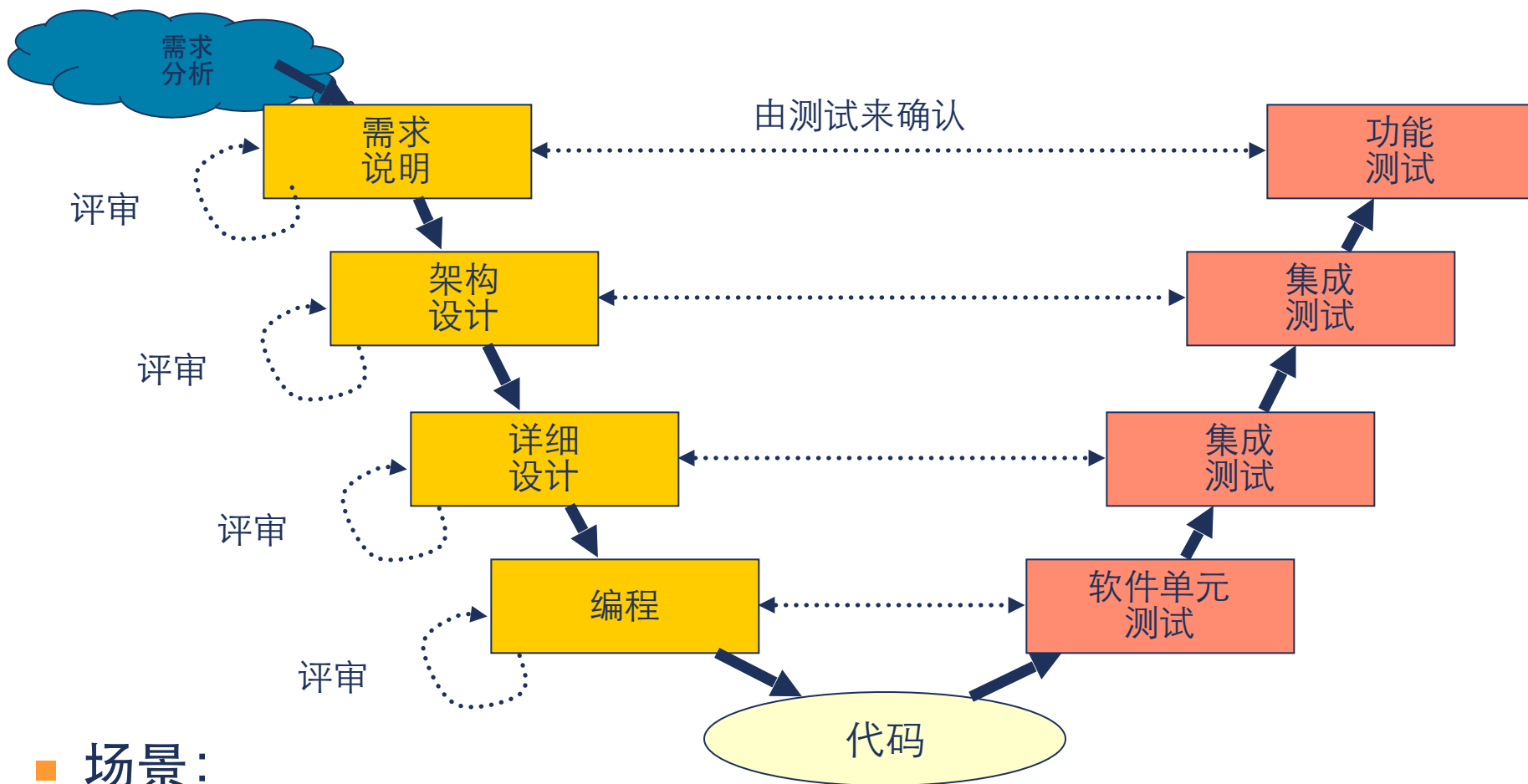


图: Wikipedia

V模式介绍



■ 场景:

- 把各个阶段前后连接起来更有效
- 实现“可追溯性”保证没有遗漏落下的

系统结构设计

- ◆ 系统如何实现设计说明书描述的功能
- ◆ 软件/硬件如何进行划分
 - 嵌入式系统中软件和硬件协同完成系统的功能
 - 软件/硬件划分通常由速度、灵活性以及开销来决策

软硬件的划分

嵌入式系统的设计涉及硬件与软件部件，设计中必须决定什么功能由硬件实现，什么功能由软件实现。

- ◆ 硬件和软件具有双重性
- ◆ 软硬件变动对系统的决策造成影响
- ◆ 划分和选择需要考虑多种因素
- ◆ 硬件和软件的双重性是划分决策的前提

通常由软件实现的部分

- 操作系统功能
 - 任务调度
 - 资源管理
 - 设备驱动
- 协议栈
 - TCP/IP
- 应用软件框架
- 除基本系统、物理接口、基本逻辑电路，许多由硬件实现的功能都可以由软件实现。

双重性部分

● 算法

● 加密 / 解密

● 编码 / 解码

● 压缩 / 解压

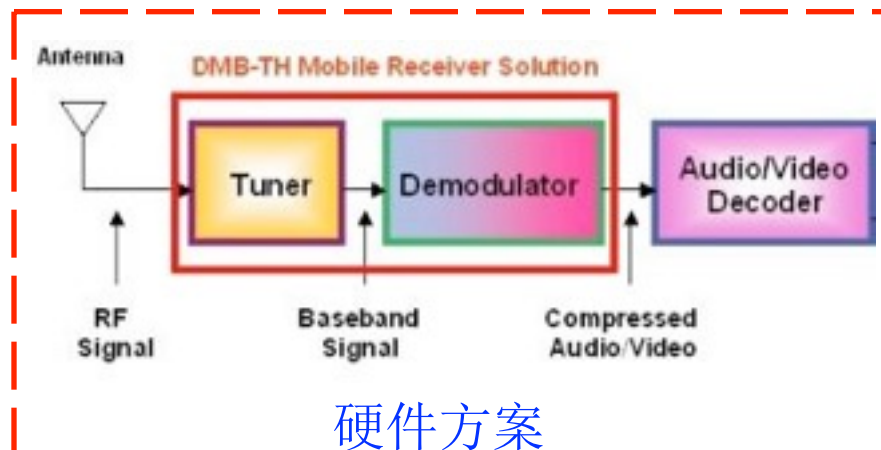
●

● 数学运算

● 浮点运算, **FFT**, ...

●

例如：移动电视的条件接收系统

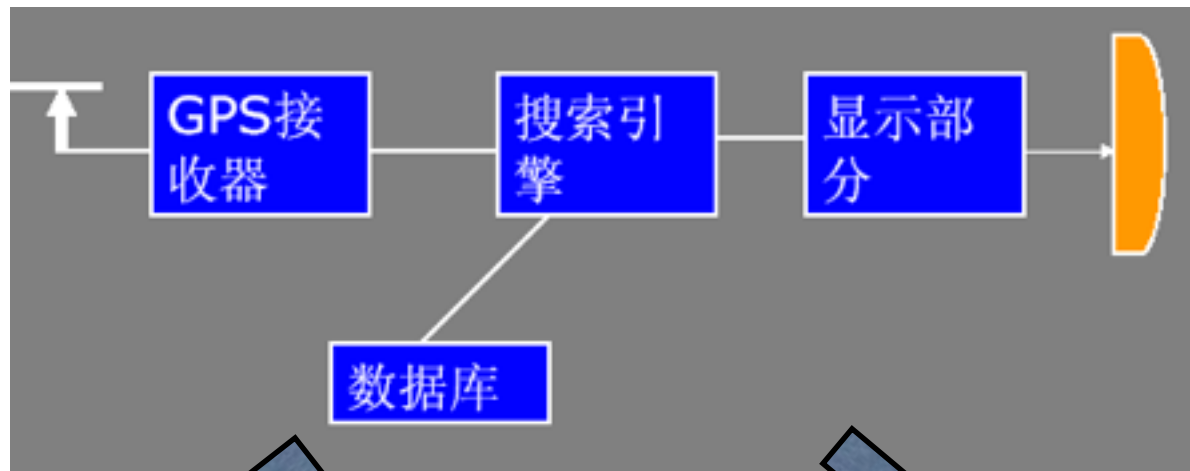


硬件方案

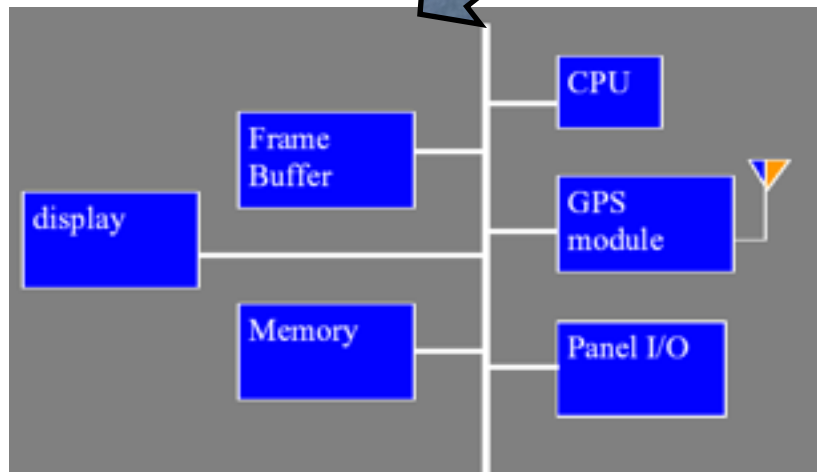
- CA数据缓存区开辟与处理
- 当前终端信息的获取
- 向Smartcard发送APDU指令, 完成与智能卡的通讯

软件方案

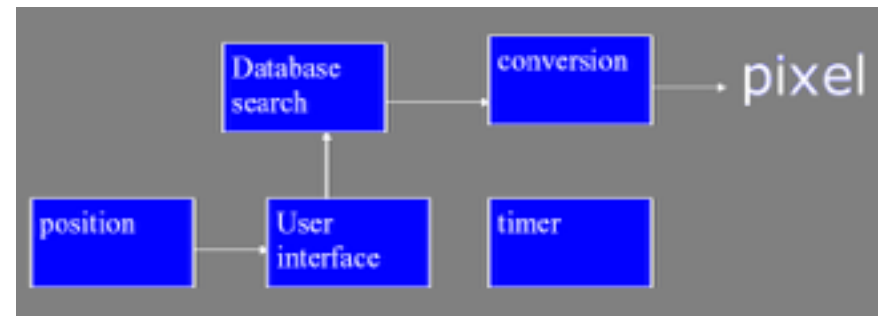
例：GPS移动地图的结构设计



硬件结构



软件结构

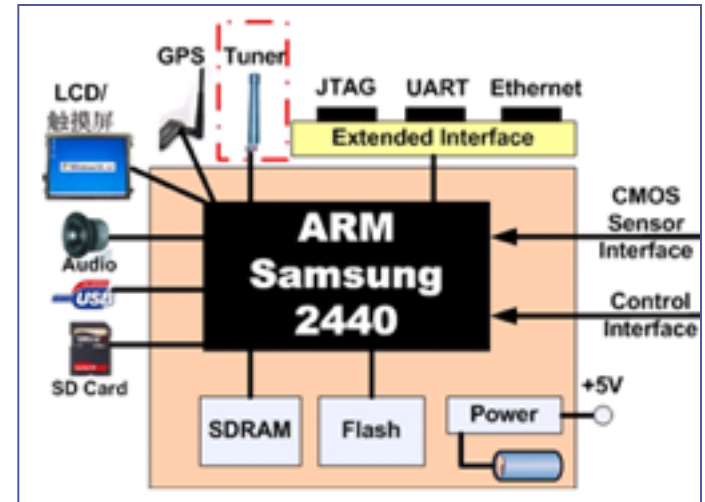


标准构件和自行设计构件

- 构件的实现
 - 选择标准
 - 自行设计
- 标准构件
 - 已经产品化
 - 形成规模生产
- 标准构件 + 自行设计构件 = 用户系统
- 构件包括了硬件构件和软件构件
- 构件本身可以是层次性的，可以由子构件组成

标准硬构件

- 硬构件的形式：
 - **IC**: 集成电路
 - **PCB**: 印刷电路板
 - **IP: Intellectual Property**
 - 标准 **IC**
 - **CPU, DSP,**
 - **RAM, ROM, 接口控制器,**
 - **ASIC,**
 - 标准 **IP**
 - **CPU核,**
 - 标准模块



例：GPS移动地图硬件部分

标准软构件

- **OS / RTOS**

- 协议栈

- **TCP/IP**

- 路由协议

- **H.323**

-

- 图形开发包

- **VxWorks**的**ZINK**

-

- 驱动程序



例：GPS移动地图软件部分

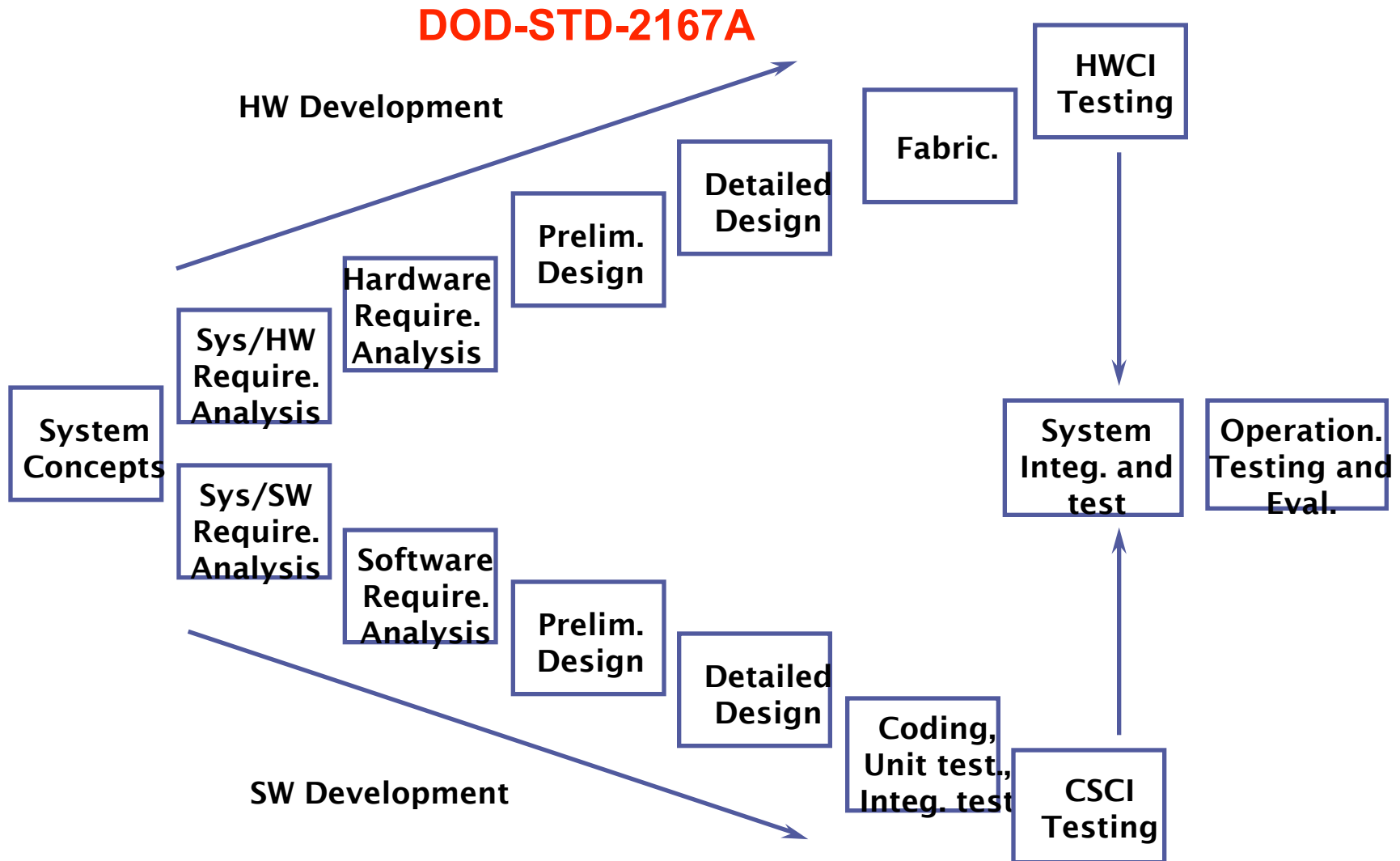
自研硬构件

- 内容
 - 逻辑电路，专用加速器，
- 实现方式
 - **PCB:**
 - **IC: PLD FPGA ASIC,**
- **EDA**设计工具
 - 板级：
 - 原理图设计工具
 - **PCB**设计工具
 - **IC**

自研软构件

- **BSP**
- 驱动程序
- 应用程序
-

传统的嵌入式系统设计模型



传统的嵌入式系统设计过程

◆ 传统软硬件设计过程的基本特征：

- 系统在设计一开始就被划分为软件和硬件两大部分
- 软件和硬件独立进行开发设计
- **“Hardware first” approach often adopted**

◆ 隐含的一些问题：

- 软硬件之间的交互受到很大限制，软硬件之间的相互性能影响很难评估
- 系统集成相对滞后，**NRE**（**Non-Recurring Engineering cost**）大

◆ 因此：

- 设计修改困难、研制周期无法有效保障

传统设计过程中的尖锐矛盾

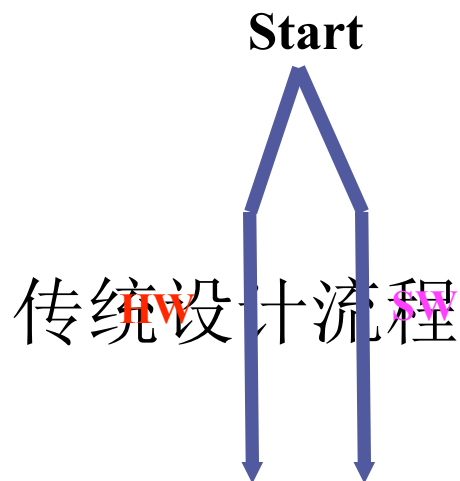
- ◆ 随着设计复杂程度的提高，软硬件设计中的一些错误将使开发过程付出昂贵的代价
- ◆ “**Hardware first**” approach often compounds (混合) software cost because software must compensate for (补偿) hardware inadequacies (不充分)

软硬件技术发展对嵌入式系统设计的影响

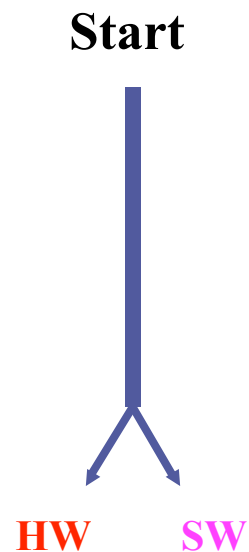
- 软硬件设计的趋势——融合、渗透
 - 硬件设计的软件化
 - VHDL, Verilog
 - HANDL-C
 - 软件实现的硬件化
 - 各种算法的ASIC
- 对系统设计的影响——协同设计
 - 增加灵活性
 - 增加了风险

软硬件设计过程发展方向—协同设计

协同设计流程



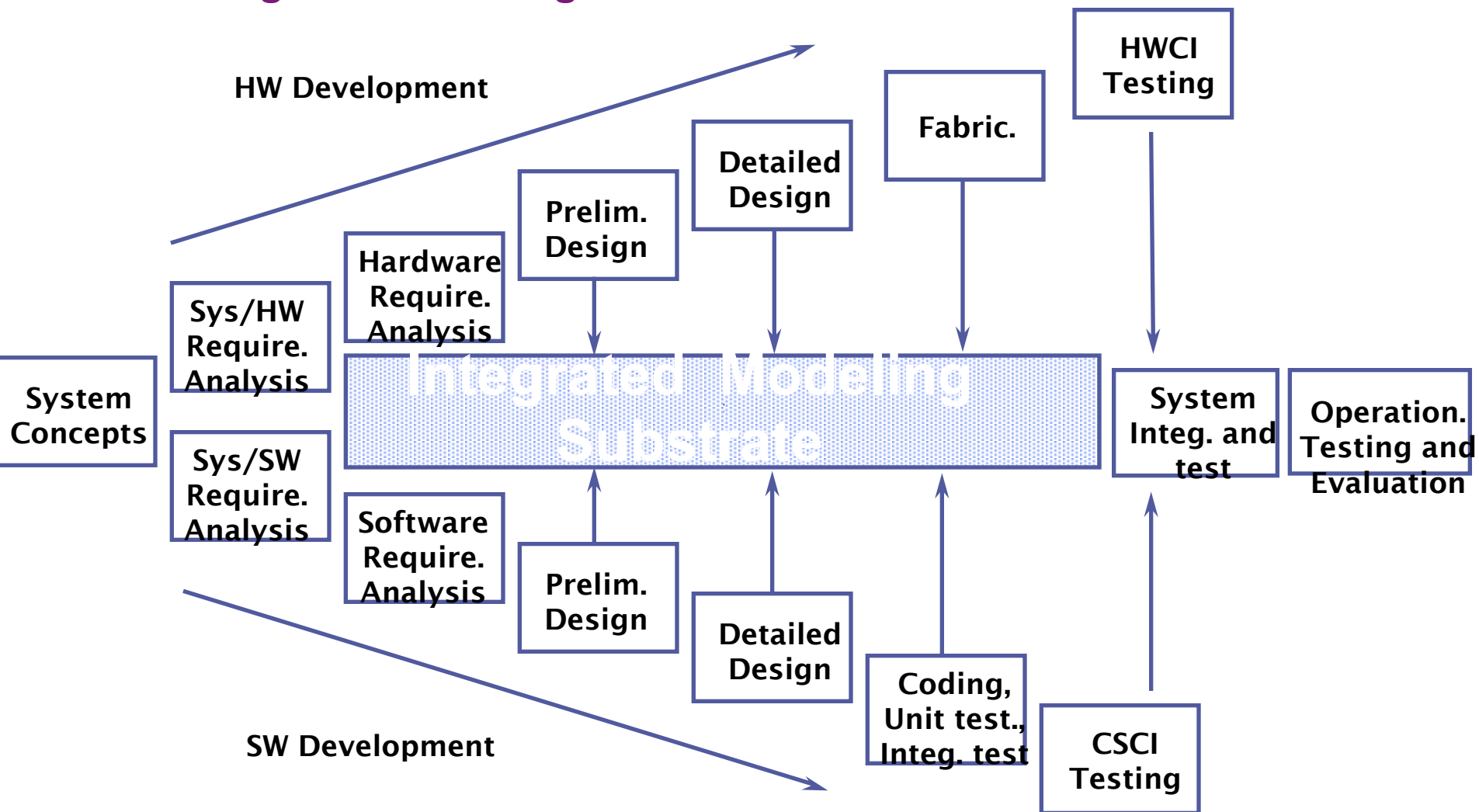
**Designed by independent
groups of experts**



**Designed by Same group of
experts with cooperation**

软硬件设计过程发展方向—协同设计

Integrated Modeling Substrate (一体化建模底层)



软件硬件协同设计的设计流程

- ◆ 用HDL语言和C语言进行系统描述并进行模拟仿真和系统功能验证；
- ◆ 对软硬件实现进行功能划分，分别用语言进行设计并将其综合起来进行功能验证和性能预测等仿真确认(协调模拟仿真)；
- ◆ 如无问题则进行软件和硬件详细设计；
- ◆ 最后进行系统测试。

4. 编程和代码检查

- 编程是由详细设计说明直接驱动的
- 开发代码的时候使用版本控制系统
- 遵循编程标准
 - 消除或导致代码理解困难的文体变异（个人风格、古怪写法）
 - 避免已知的会导致误解的做法
 - 详细说明最好的做法，使其易于遵照执行
- 做代码评审
- 有效的测试
 - 测试自动化
 - 回归测试

捉对代码评审

- 在测试代码之前先做评审
- 代码检查给企业带来了大量的正面效果
 - IBM从中找出了80%的错误
 - 每发现一个错误可以节省9个小时
 - AT&T发现质量提升了1000%而产能提升了14%
- 发现测试往往会遗漏的错误
 - HP的检查发现的错误中，有80%不可能被测试捉到
 - HP、Shell Research、Bell Northern和AT&T：检查比测试有效20-30倍

5. 软件测试

- 测试不是“验证程序正确工作的过程”
 - 程序可能不能在所有可能的情况下正确工作
 - 专业的程序员在“写好了”程序之后，每100行代码会有1到3个错误
- 测试人员不应该试图证明程序正确工作（不可能）
 - 如果你想，而且期望你的程序能工作，就会无意识地遗漏可能的失败，因为人类天生就不能做到不偏不倚
- 测试的目的是快速找到问题
 - 软件是否不符合定义？
 - 软件是否不符合未声明的需求？
- 找到问题的目的是修复有影响的问题
 - 修复最重要的问题，因为没有足够的时间来修复所有的问题
 - 帕累托法则（80/20原则）定义了“重要的少数，琐碎的多数”
 - 错误的频度是不一致的——重要的少数导致了程序失效的主要原因，首先要处理这些

测试的途径

■ 增量测试

- 写一个函数就测试（模块/单元/码元测试）
- 然后把几个有关系的函数放在一起测试（集成测试）
 - 随着写出新的函数，不断扩展测试的范围
- 增量测试需要额外的代码来做测试工具
 - 用驱动函数来调用被测函数
 - 可能需要用模拟函数来仿真被测函数调用的那些会返回或修改数据的函数
 - 测试工具可以自动测试单个函数来发现今后还会出现错误

■ 突击测试

- 编程做好所有的函数来构建出整个系统
- 测试完整的系统
 - 启动然后祈祷

为何做增量测试？

■ 找出什么失败了要容易很多

- 突击的话，因为没有函数被彻底测试过，大多数可能带有错误
- 那么问题就成了“我看到的失败到底是哪个模块里的哪个错误造成的？”
- 一个模块中的错误可能让其他模块的测试难以实施
 - 在基础模块（比如内核）中的错误可以在很多依赖于它的模块中表现出来

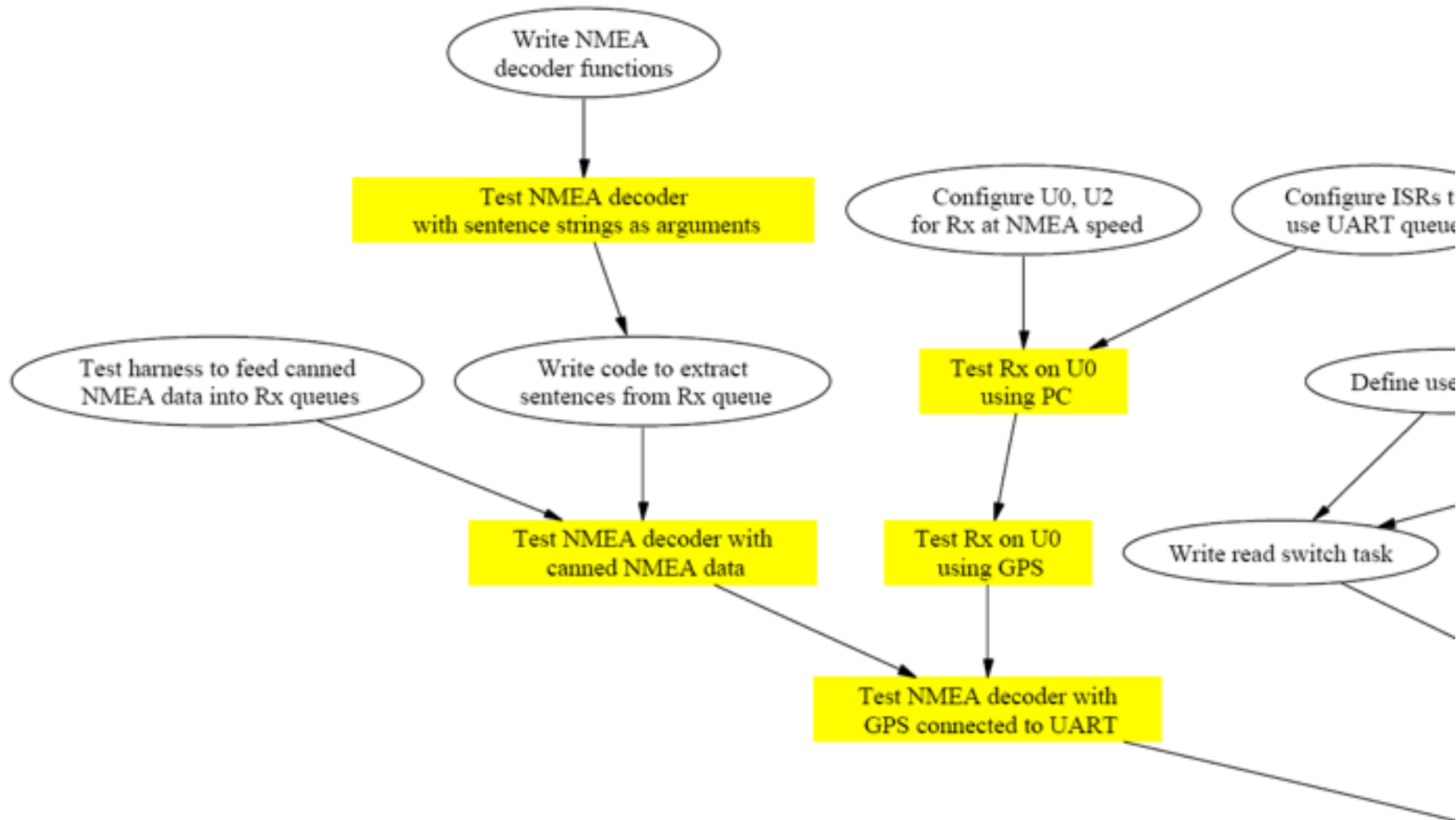
■ 更少的互相指责 = 更幸福的软件开发团队

- 谁造成的错误是清楚的，谁需要修改也是清晰的

■ 更好的自动化

- 驱动函数和模拟函数一开始需要花时间来编写，但是在将来的测试中能节省时间

测试计划的例子



6. 做项目总结

- 目标——改进工程过程
 - 从刚刚完成的项目中吸取一切有用的信息——给其他人提供“虚拟经验”
 - 提供正面的非对抗性的反馈
 - 清晰地记录问题和解决方案用于今后的工作中
- 基本规则：有问题就需要解决
- 往往小的改变能提升性能，但是也容易被忘记

事后分析结构的例子

■ 产品

- 错误
- 软件设计
- 硬件设计

■ 过程

- 代码标准
- 代码接口
- 变更控制
- 我们是如何做的
- 团队协作

■ 工具

- 燃尽图
- 变更单
- 个人可用性

■ 支持

课程大纲

 嵌入式处理器的补充介绍

 嵌入式系统开发方法概述

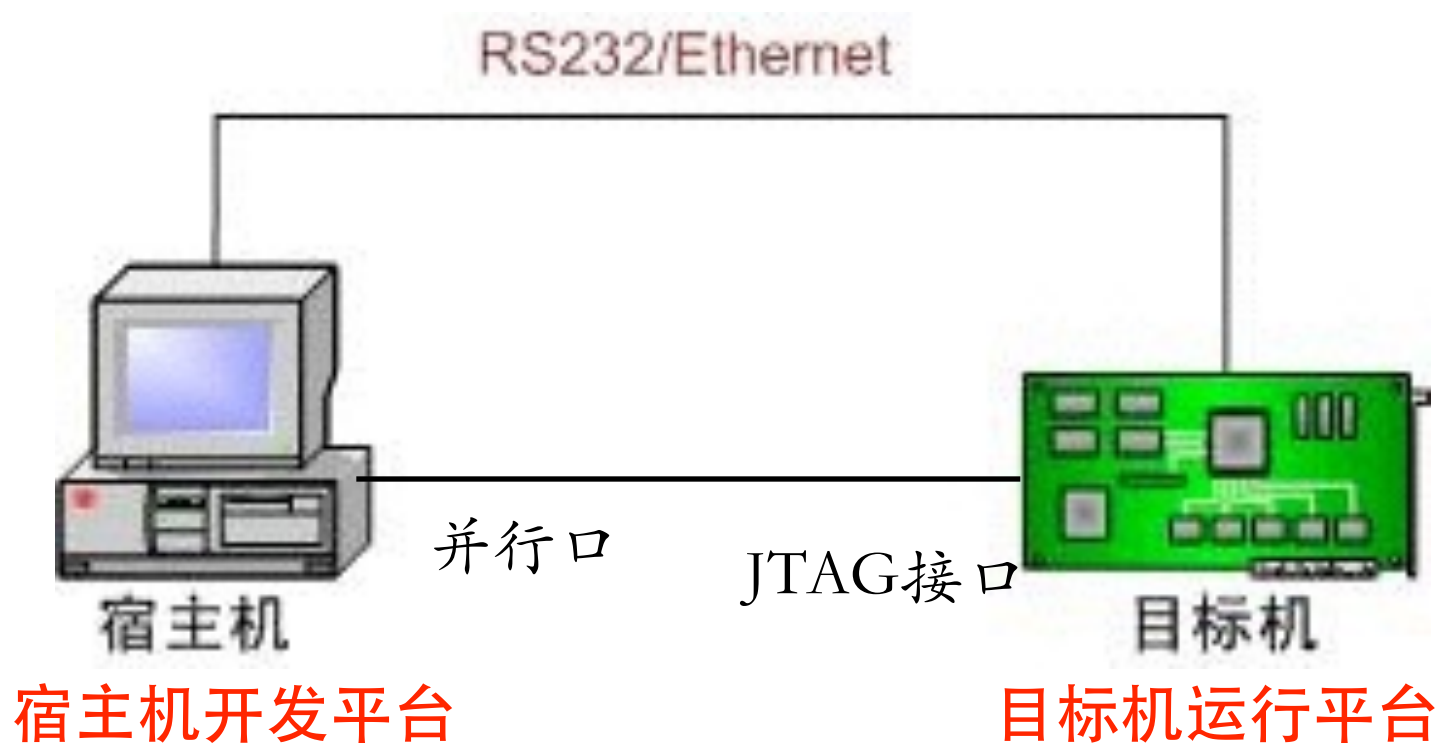
 嵌入式系统软件开发概述

 Linux操作系统、虚拟机概述

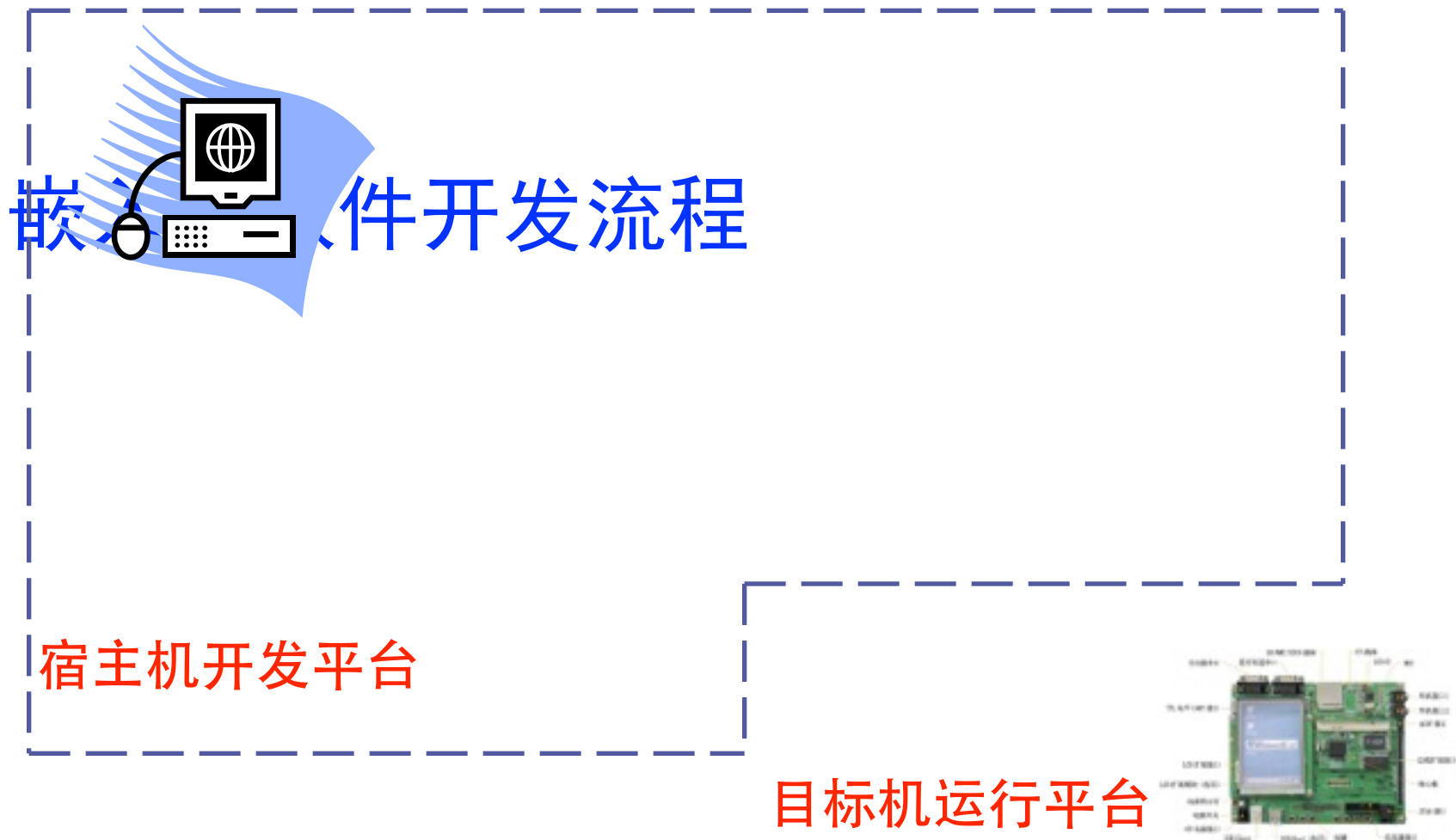
 GNU开发工具链简介

 嵌入式软件开发环境建立实验

宿主机—目标机开发模式

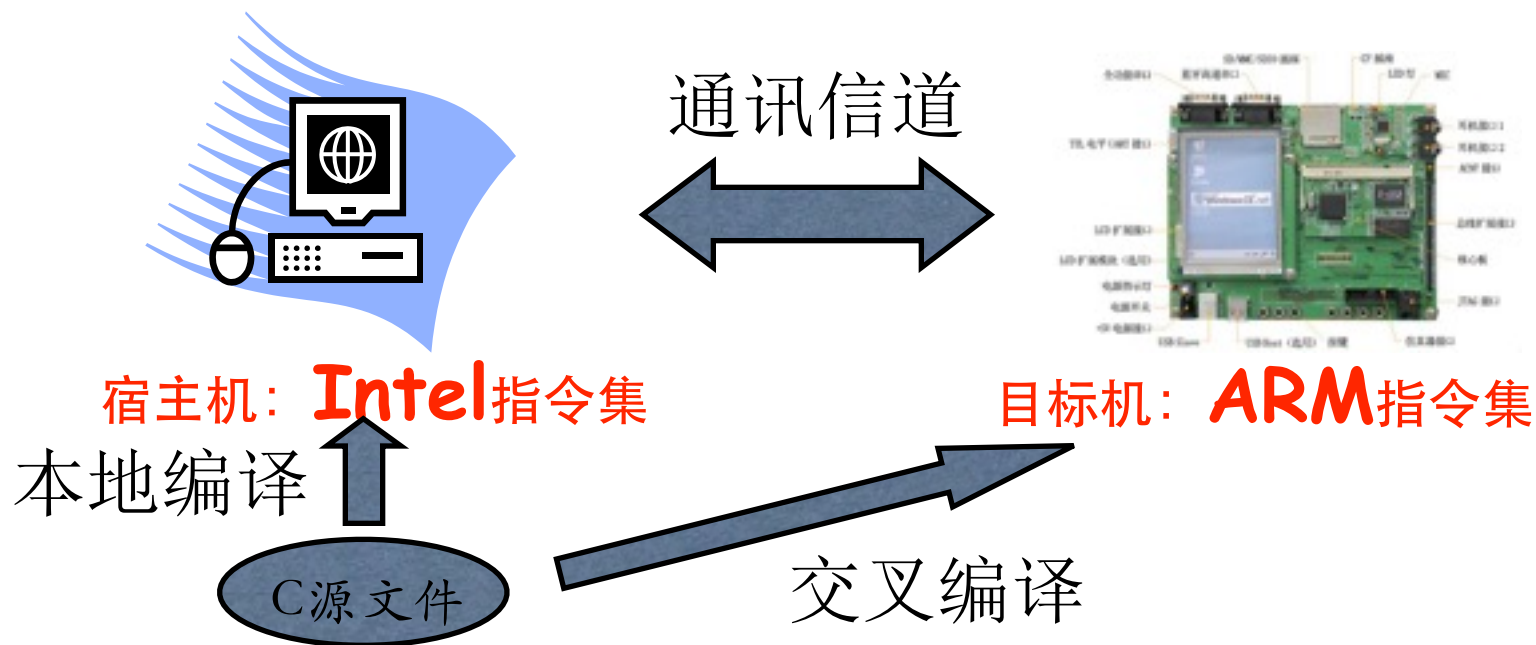


嵌入式软件开发流程



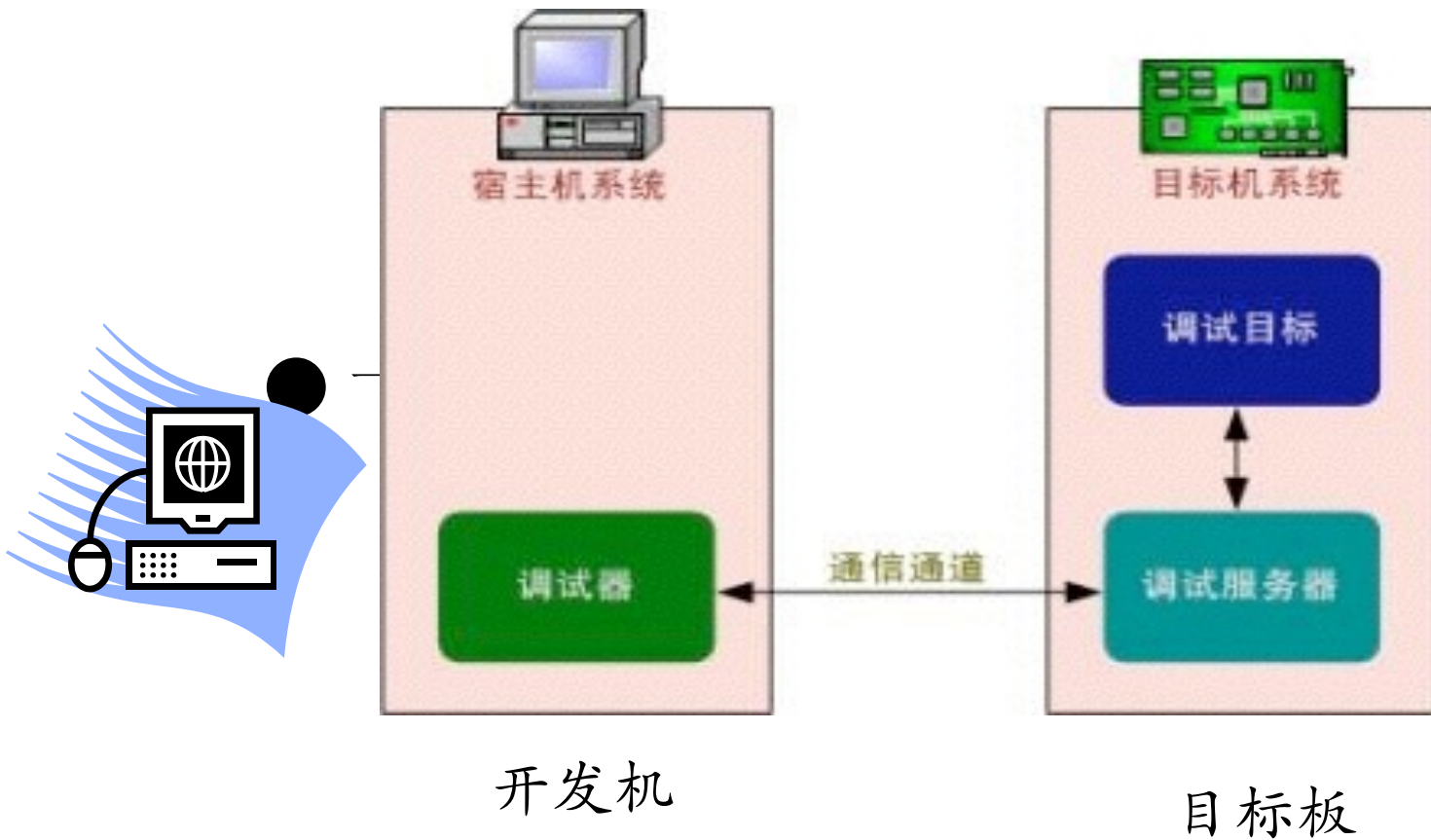
交叉编译 VS 本地编译

- 交叉编译器和交叉链接器是指能够在宿主机上安装，但是能够生成在目标机上直接运行的二进制代码的编译器和链接器

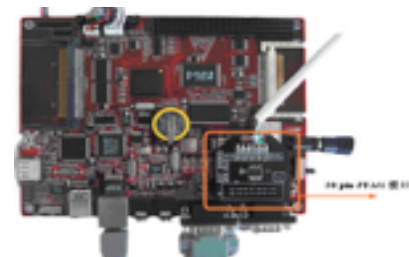


- 基于ARM体系结构的gcc交叉开发环境中，arm-linux-gcc是交叉编译器，arm-linux-ld是交叉链接器

远程调试概述



图：

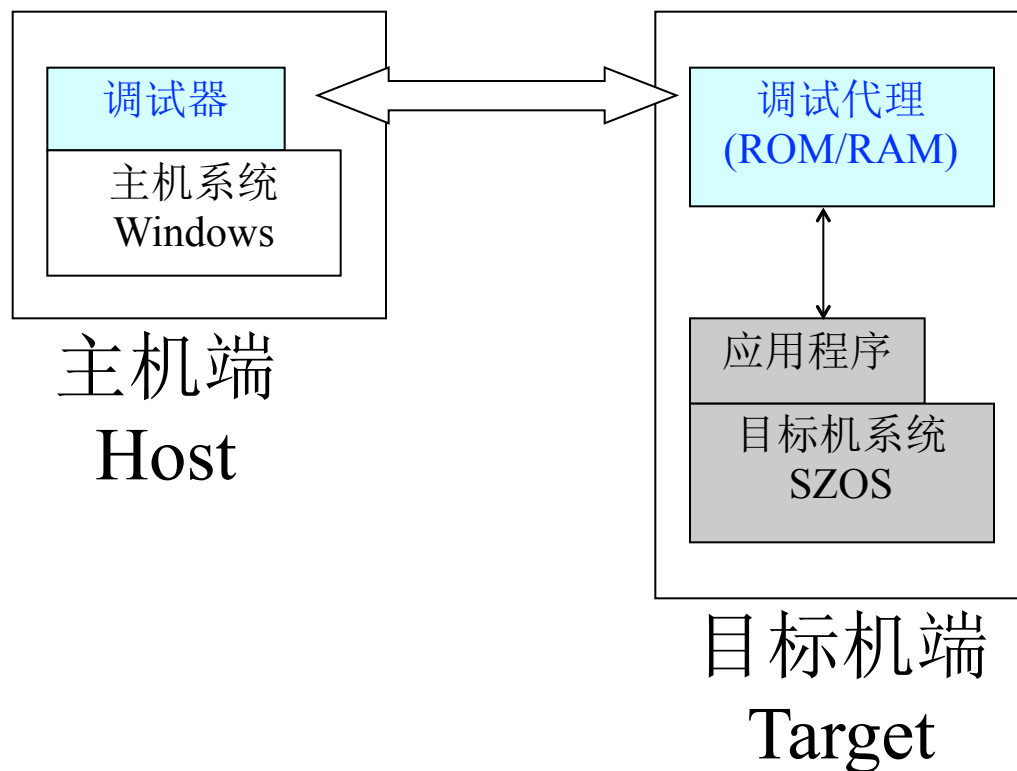


远程调试特点

- 嵌入式系统远程调试方法有很多，但一般都具有以下特点：
 - 调试器和被调试进程运行在不同的机器上
 - 调试器通过某种通信方式与被调试进程建立联系
 - 在目标机上一般会具备某种形式的调试代理
 - 目标机可能是某种形式的系统仿真器

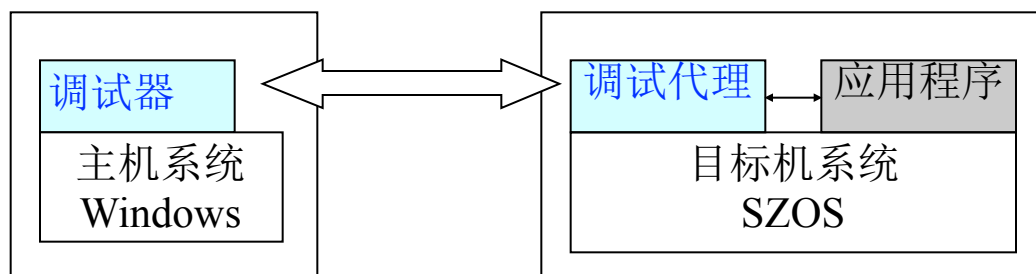
远程调试基本原理—系统级调试

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



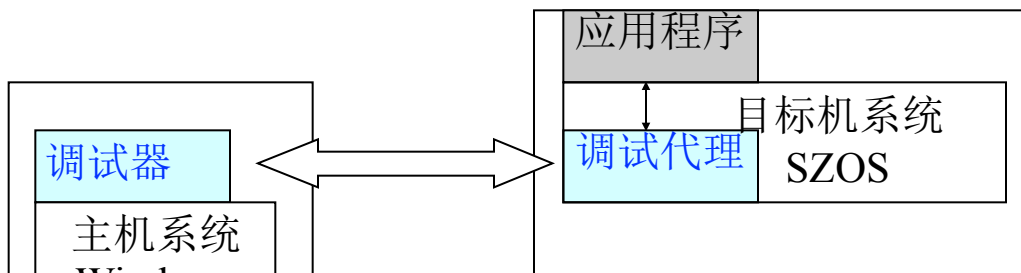
远程调试基本原理—任务级调试

根据调试代理和实时操作系统的关系，任务级调试模型又可以分为两种：



主机端Host

目标机端Target



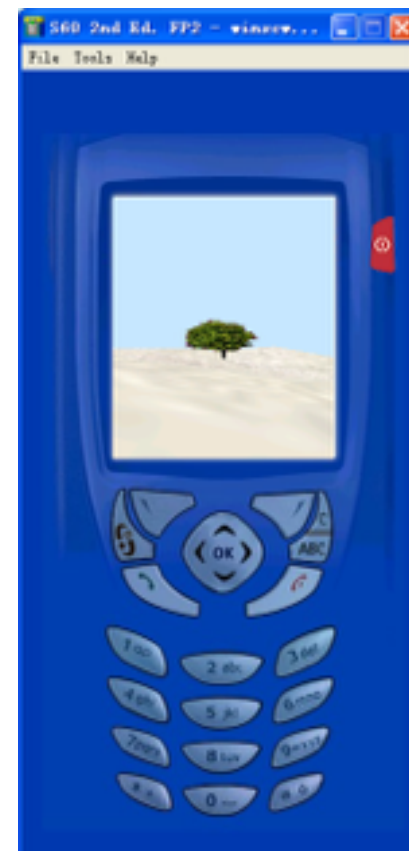
目标机端Target

主机端Host

仿真调试

◆ 非目标机平台仿真运行、调试与测试

- 源程序模拟器 (**Simulator**)
- 实时在线仿真系统**ICE** (**In-Circuit Emulator**)



源程序模拟器（**Simulator**）

- ◆ 以一种处理器，模拟另一种处理器的运行
- ◆ 简单的模拟器
 - 通过指令解释方式逐条执行源程序
 - 分配虚拟存储空间和外设
 - 供程序员检查
- ◆ 高级的模拟器（全系统模拟）
 - 模拟外部设备的**I/O**电气信号

源程序模拟器举例



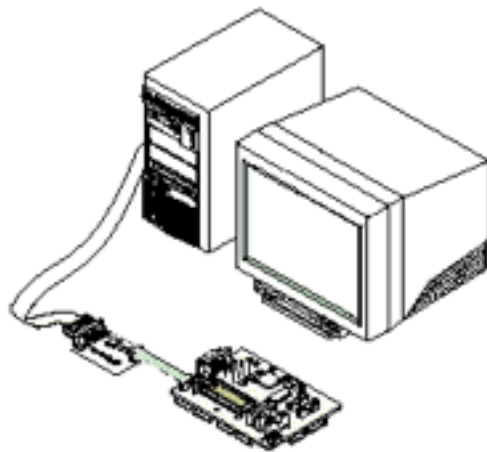
x86模拟器Virtual Box



Android模拟器

实时在线仿真**ICE** (**In-Circuit Emulator**)

- **ICE**具有的功能
 - 排除设计逻辑错误
 - 发现与排除硬件干扰等引起的异常执行行为
 - 高级的**ICE**带有完善的跟踪功能
 - 同时也是提高和优化系统性能指标的工具



集成开发环境

- 随着实时操作系统的使用，集成开发环境（IDE）也越来越被开发人员所需要。
- 一个好的嵌入式系统开发环境将能更好地发挥实时操作系统优势，并为整个嵌入式应用程序的开发奠定良好的基石。

集成开发环境功能

模块分类	子模块分类	功能描述	模块分类	子模块分类	功能描述	必备功能	工作基础
编译环境	C/C++编译器	调用其他程序实现将程序源文件编译成目标文件的功能	项目管理器	项目管理器	负责应用源程序的文件组织、管理	*	
	汇编器	汇编程序转换为 ELF 格式的目标文件		版本管理器	对源程序版本进行组织与管理		
	链接器	链接目标程序		查询辅助工具	对源程序中文本提供模式查询、匹配	*	
	库管理器	将多个目标文件归档为一个函数库文件	配置工具	选项配置工具	配置编译链接选项、目标机 OS 参数等	*	
	工程管理器	自动编译、链接的配置工具		目标代码剪裁	对目标代码提供可视化裁减功能	*	*
			辅助分析与诊断工具	覆盖测试工具	分析测试代码的覆盖率执行时间等		
交叉调试环境	交叉调试器 (Host 端)	交叉调试器的 Host 端，给 Target 端发送调试命令		单元测试工具	提供模块单元测试功能		
	目标监控器 (Target 端)	交叉调试器的 Target 端，接受 Host 端命令并实现断点等调试功能		内存调试工具	查看和修改内存信息，包括寄存器信息	*	
	目标码下载	将目标代码由开发机向目标机下载		对象浏览工具	查看目标机中各类事件、信号对象，以及多任务通讯、同步、阻塞、抢占情况	*	
源码编辑器	源码编辑器	支持源码编辑和语法色彩等附加功能		性能调试工具	查看和比较系统性能		
				系统仿真工具	仿真目标机运行环境		
				监视分析工具	提供系统运行对象监视及查询分析	*	
				Shell	命令解释工具	*	

键入文本

集成开发环境技术发展趋势

采用Eclipse开源框架作为集成开发环境基础，已经应用得较为广泛，例如：

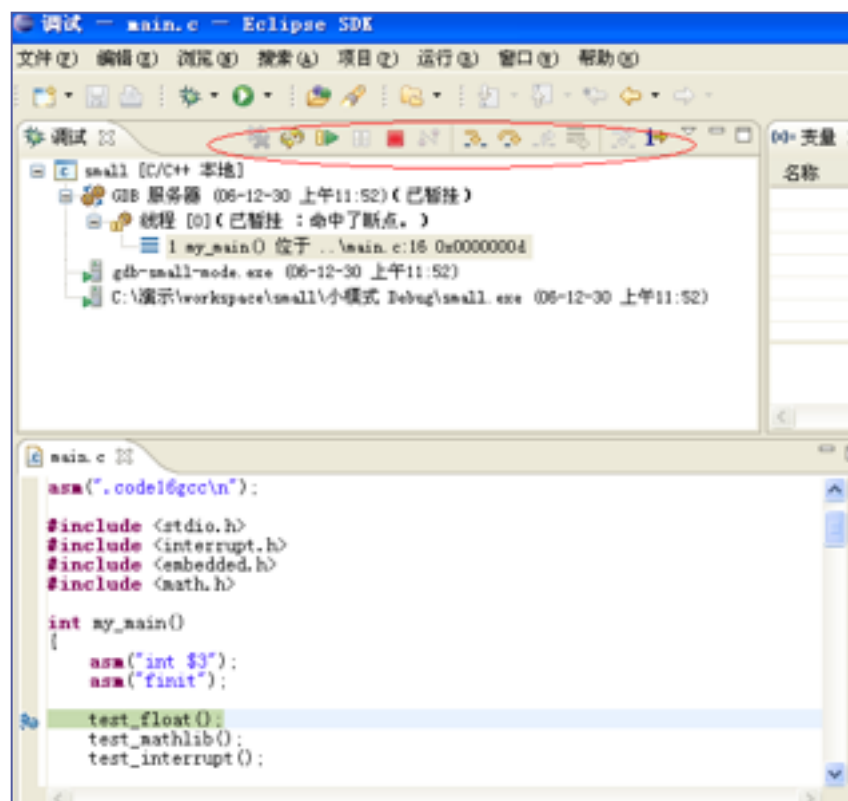
- WindRiver公司在VxWorks6.x版本中推出的Workbench
- 美国LynuxWorks公司的Luminosity
- 加拿大QNX公司推出的Momentics
- 美国TimeSys公司的TimeStorm

均基于Eclipse框架的集成开发环境

开发工具链概述

在集成开发环境中提供支持嵌入式系统开发的工具链，最核心的包括：

- 编译器
- 链接器
- 开发库
- 汇编器
- 调试器
- 反汇编器
- 可执行代码抽取



课程大纲

 嵌入式处理器的补充介绍

 嵌入式系统开发方法概述

 嵌入式系统软件开发概述

 Linux操作系统、虚拟机概述

 GNU开发工具链简介

 嵌入式软件开发环境建立实验

概述

- **Linux**操作系统简介
- 虚拟机概述
- **Ubuntu Linux**常用命令简介
- **Linux**文本编辑器简介

Ubuntu Linux常用命令简介

□ 文件系统命令

命令	说明	命令	说明
pwd	显示当前目录	cd	进入目录
ls (dir)	显示当前目录内容	mkdir	创建目录
rmdir	删除目录	rm(del)	删除文件
mv	移动文件	cp	拷贝文件
cat	打印文件内容	mount	挂载文件系统

□ 任务操作命令

命令	说明	命令	说明
ps	显示当前任务	kill	删除指定任务

Ubuntu Linux常用命令简介

□ 其他相关命令

命令	说明	命令	说明
minicom	启动、配置串口命令	ifconfig	网络配置命令
tar	文件打包/释放命令	man	命令帮助显示

Ubuntu Linux文本编辑简介

□ Vi (visual interface)

Vi是**Linux**中最常用的全屏编辑器，所有**Linux**系统中均提供该编辑器。只要操作习惯，比任何编辑器功能都强大。

□ Emacs

Emacs是另一个功能强大的全屏编辑器。

虚拟机应用背景：计算系统灵活性不高

作业：飞行器的
流场计算

作业：生物计
算



飞行器专家

程序：**Fluent**

6.3

作业：**32位**

操作系统：



生物专家

程序：

BLAST

作业：**32位**

操作系统：

- ◆ 浪费时间
- ◆ 操作繁琐
- ◆ 机器闲置时间较多

Linux

平台

X86



操作系统不匹配！

操作系统不匹配！

平台：**X86**

◆ 原因：

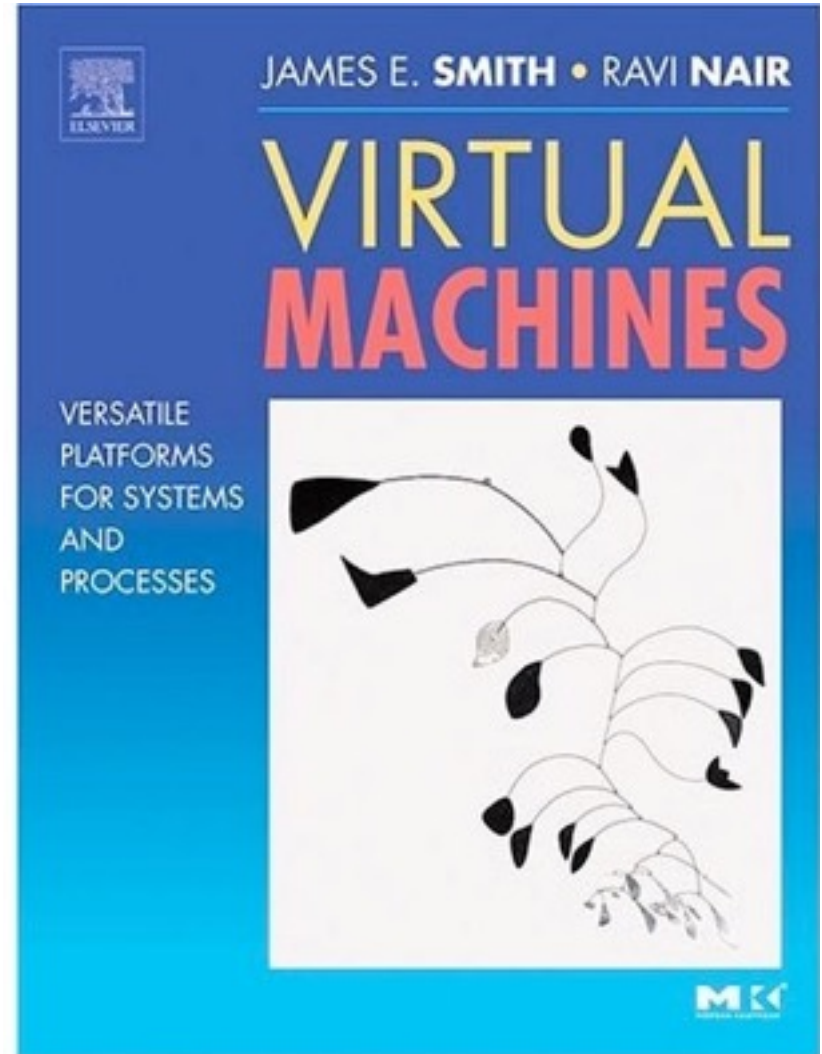
- 应用软件与操作系统、硬件紧耦合



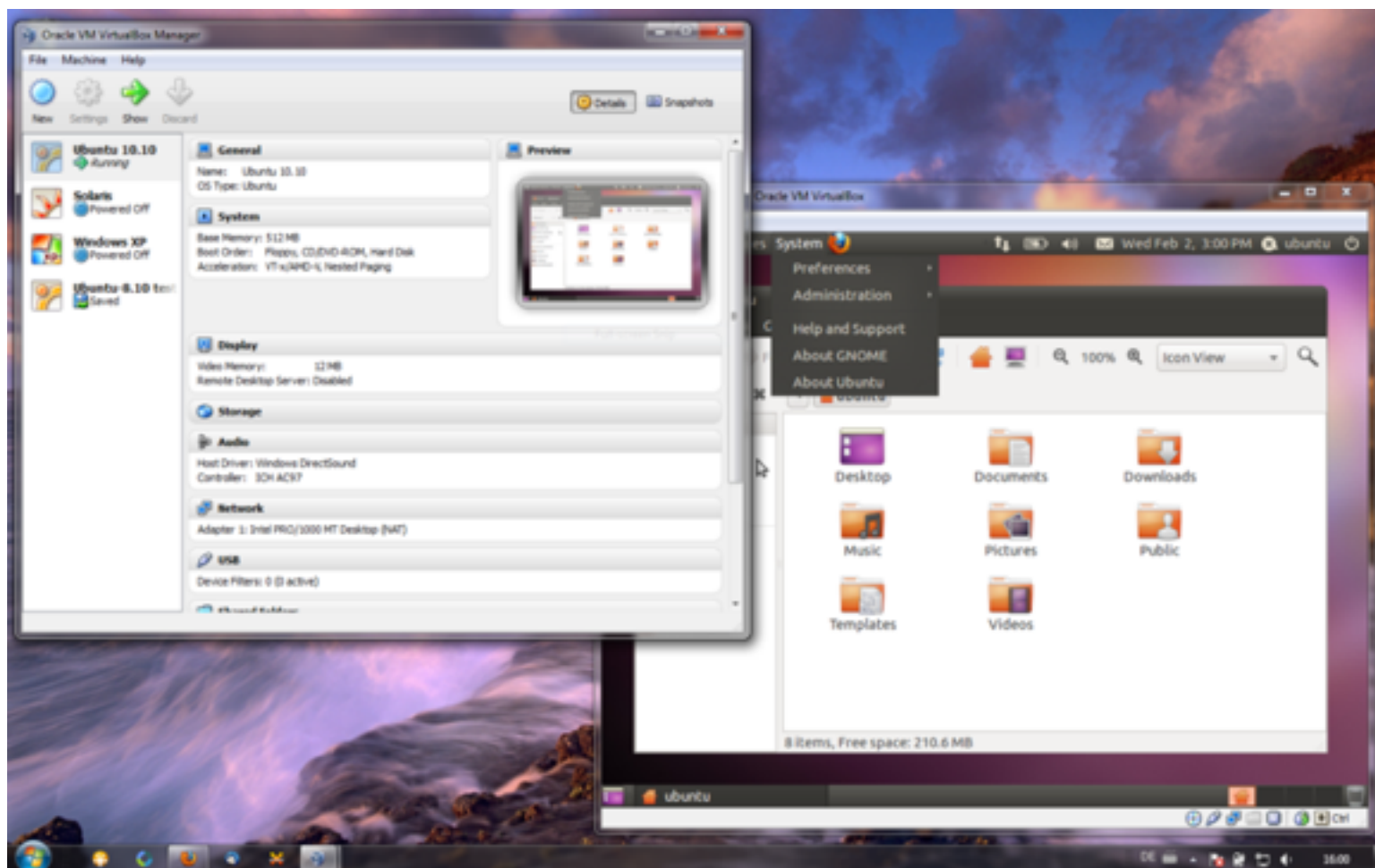
计算中心

虚拟化方法：去耦合

- 虚拟化是将底层物理设备与上层操作系统、软件分离的一种去耦合技术
- 虚拟化的目标是实现IT资源利用效率和灵活性的最大化



虚拟机



课程大纲

 嵌入式处理器的补充介绍

 嵌入式系统开发方法概述

 嵌入式系统软件开发概述

 Linux操作系统、虚拟机概述

 GNU开发工具链简介

 嵌入式软件开发环境建立实验

GNU工具链简介

- **GNU**工具链提供了一系列开源的开发工具，有效地支持了嵌入式软件的开发，包括：编译器：GCC、链接器：LD、汇编器：GAS、开发库：GLIBC、调试器：GDB、反汇编器：OBJDUMP、可执行代码抽取工具：OBJCOPY、工程管理系统：MAKE。
- **GNU**工具链是目前开源开发工具套件领域中，应用最为广泛、最为稳定的开发工具链。

GNU binutils二进制工具集

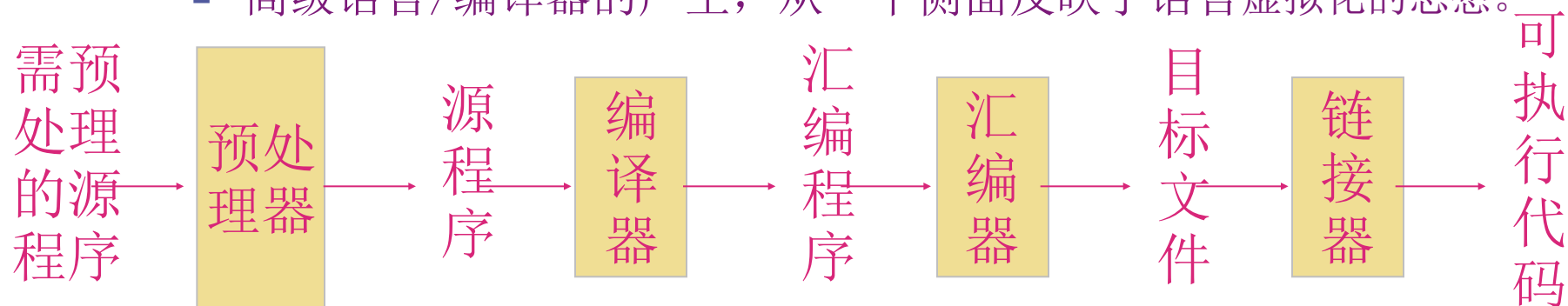
- ❑ **ar**: 库文件维护工具。
- ❑ **ranlib**: 创建、添加库文件索引。
- ❑ **nm**: 列出目标文件符号表。
- ❑ **objcopy**: 目标文件格式转换。
- ❑ **objdump**: 反汇编工具。
- ❑ **strip**: 从目标文件或文档库中去掉符号表等调试信息。
- ❑ **addr2line**: 将可执行文件内部地址转换成源代码文件名和行号。
- ❑ **size**: 列出目标文件中每个部分的名字和尺寸。
- ❑ **gprof**: 与编译程序配合，报告程序中各个函数的运行时间。
- ❑ **strings**: 浏览所有类型的文件，析取出用于显示的字符串。

什么是编译器

□ **编译器**：一个语言翻译程序（如GNU GCC），负责将用高级语言（c/c++等）编写的源程序，转换成等价的低级语言的翻译程序。

□ **编译器作用**：

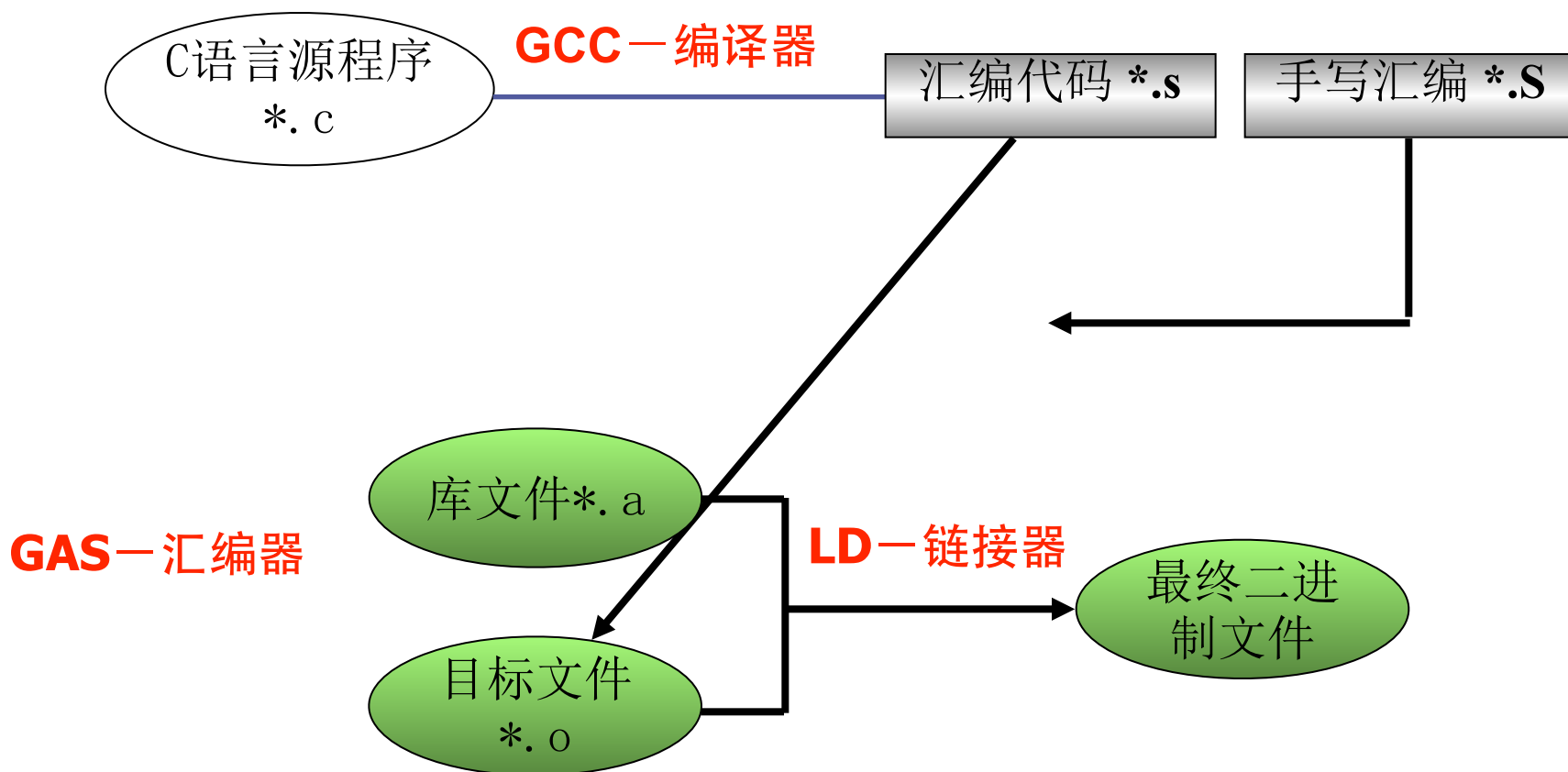
- 使得程序员不必考虑与机器相关的繁琐细节，从而独立于机器。
- 高级语言/编译器的产生，从一个侧面反映了语言虚拟化的思想。



三个与编译相关的问题

- 编译的过程是怎样完成的？
- 目标文件的组成及加载方式？
- 目标文件与可执行代码有何区别？

编译过程概述——以GCC为例



GCC编译命令

■GCC语法: `gcc [选项] 文件列表`

■常用选项

■**-o**: 文件名: 指定生成的可执行文件名, 默认是**a.out**

■**-c**: 将输入文件生成目标文件***.o**

■**-l**: 库文件名: 链接指定库文件

■**-E**: 只对源文件进行预处理, 如宏替换等

■**-S**: 将输入文件生成汇编代码***.s**

■举例:

■编译: `gcc first.c` 执行: `./a.out`

■编译: `gcc -o hello first.c` 执行: `./hello`

■编译: `gcc -c driver.c` 生成: `driver.o`

GCC预编译过程

□ `gcc -E test.c -o test.i`

或者

□ `cpp test.c > test.i`

```
#include <stdio.h>
```

```
void main()
```

```
{
```

```
char str[]="hello world!\n";
```

```
printf(str);
```

```
}
```

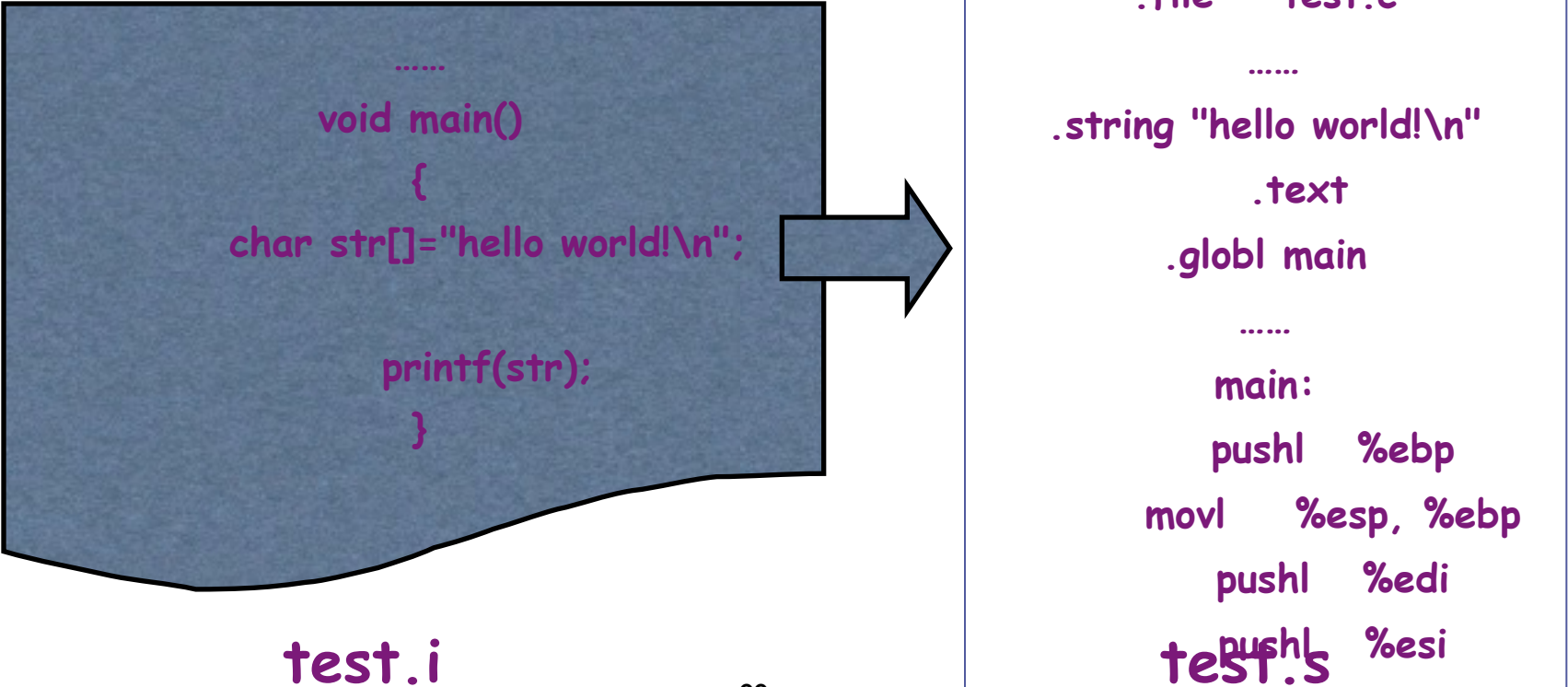
- 
- 递归展开**#include**
 - 展开**#define**宏定义
 - 删除“//”、“/**/”注释信息

GCC编译过程

□ `gcc -S test.i -o test.s`

或者

□ `/usr/lib/gcc-lib/i386-redhat-linux/3.2.2/cc1 test.c`



```
.....  
void main()  
{  
char str[]="hello world!\n";  
  
printf(str);  
}
```

test.i

```
.file "test.c"  
.....  
.string "hello world!\n"  
.text  
.globl main  
.....  
main:  
pushl %ebp  
movl %esp, %ebp  
pushl %edi  
pushl %esi  
test.s  
.....
```

GCC汇编过程

□ `gcc -c test.s -o test.o`

或者

□ `as test.s -o test.o`

```
.file "test.c"
.....
.string "hello world!\n"
.text
.globl main
.....
main:
pushl %ebp
.....
```

test.s



```
00000000 <main>:
0: 55      push %ebp
1: 89 e5    mov  %esp,%ebp
3: 57      push %edi
4: 56      push %esi
5: 83 ec 10 sub  $0x10,%esp
.....
36: c9      leave
37: c3      ret
```

test.o

GCC链接过程

□ `ld -static /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc/crtbeginT.o -L/usr/lib/gcc -L/usr/lib/ -L/lib test.o -lgcc -lgcc_eh -lc /usr/lib/crtend.o /usr/lib/crtn.o`

```
00000000 <main>:
  0: 55      push  %ebp
  1: 89 e5    mov   %esp,%ebp
  3: 57      push  %edi
  4: 56      push  %esi
  5: 83 ec 10 sub  $0x10,%esp
      .....
 36: c9      leave
 37: 63      test.o ret
```

```
00048328 <main>:
0048328: 55      push  %ebp
0048329: 89 e5    mov   %esp,%ebp
004832b: 57      push  %edi
004832c: 56      push  %esi
004832d: 83 ec 10 sub  $0x10,%esp
0048330: 83 e4 f0 and  $0xffffffff0,%esp
0048333: b8 00 00 00 00 mov  $0x0,%eax
0048338: 29 c4    sub  %eax,%esp
004833a: 8d 7d e8 lea   0xffffffffe8(%ebp),%edi
004833d: be 0c 84 04 08 mov  $0x804840c,%esi
0048342: fc      cld
0048343: b9 0e 00 00 00 mov  $0xe,%ecx
0048348: f3 a4    repz mousb %ds:(%esi),%es:(%edi)
004834a: 83 ec 0c sub  $0xc,%esp
004834d: 8d 45 e8 lea   0xffffffffe8(%ebp),%eax
0048350: 50      push  %eax
0048351: e8 12 ff ff ff call  8048268 <_init+0x38>
0048356: 83 c4 10 add  $0x10,%esp
0048359: 8d 65 f8 lea   0xfffffffff8(%ebp),%esp
004835c: 5e      pop   %esi
004835d: 5f      pop   %edi
004835e: c9      leave
004835f: c3      ret
```

a.out

目标码文件组成 (1/3)

C code with various
storage classes

问题2:

全局变量定义如下

static int x2 = 1;

static int x1 = 0;

x1和**x2**会被放在什么段中?

/* some code */

static int static_var = 85;

static int static_var2;

int a = 1;
int b;

/* some code */
return 0;
}

问题1: 存储空间
分配在哪里?

.bss section

未初始化
数据段
(Block Started
by Symbol)

代码段

数据段

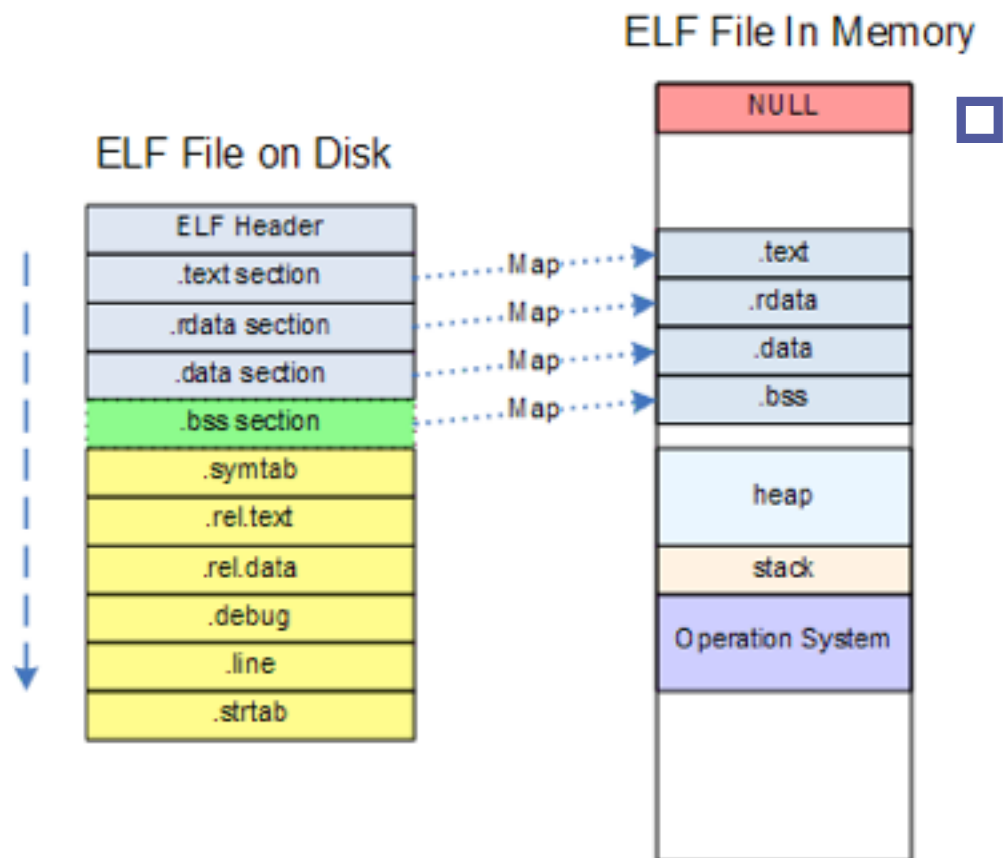
目标码文件组成（2/3）

.text	程序代码
.data	初始化的全局变量和静态变量
.bss	未初始化的全局变量和静态变量
.ctor	全局对象和静态对象的构造函数
.dtor	全局对象和静态对象的析构函数

目标码文件组成（3/3）

.rdata	read only data. 一般用于存放只读数据，比 const 型的全局初始化的变量、程序中的常量字符串等。
.symtab	symbol table. 符号表，用于存放程序中的变量名、函数名等符号。
.rel.text	relocation text. 代码段的重定位信息，用于代码段的符号重定位。
.rel.data	relocation data. 数据段的重定位信息，用于数据段的符号重定位。
.debug	debug. 调试信息，用于配合调试器（比如 gdb ）保存的一些关于调试的信息。
.line	line. 行号，用于保存源程序中的行号。
.strtab	string table. 字符串表，符号表的符号的字符串等一般都保存在 .strtab 中。

可执行文件的加载过程



加载过程

- 读入可执行文件头部信息
- 建立**text**、**data**等段
- 将**bss**段清零
- 建立堆**stack**、堆栈**stack**
- 建立程序参数、环境变量
- 将程序的起始地址指向程序入口**entry**，启动运行

目标码文件的内容

□ `objdump -d test.o`

浮动地址

```
00000000 <main>:
 0: 55                push    %ebp
 1: 89 e5             mov     %esp,%ebp
 3: 57                push    %edi
 4: 56                push    %esi
 5: 83 ec 10          sub     $0x10,%esp
 8: 83 e4 f0          and     $0xffffffff0,%esp
 b: b8 00 00 00 00    mov     $0x0,%eax
10: 29 c4             sub     %eax,%esp
12: 8d 7d e8          lea     0xffffffffe8(%ebp),%edi
15: be 00 00 00 00    mov     $0x0,%esi
1a: fc              cld
1b: b9 0e 00 00 00    mov     $0xe,%ecx
20: f3 a4             repz    movsb %ds:(%esi),%es:(%edi)
22: 83 ec 0c          sub     $0xc,%esp
25: 8d 45 e8          lea     0xffffffffe8(%ebp),%eax
28: 50                push    %eax
29: e8 fc ff ff ff    call    2a <main+0x2a>
2e: 83 c4 10          add     $0x10,%esp
31: 8d 65 f8          lea     0xfffffffff8(%ebp),%esp
34: 5e                pop     %esi
35: 5f                pop     %edi
36: c9                leave
37: c3                ret
```

重定位指令

`test.o`

可执行文件的内容

□ `objdump -d a.out`

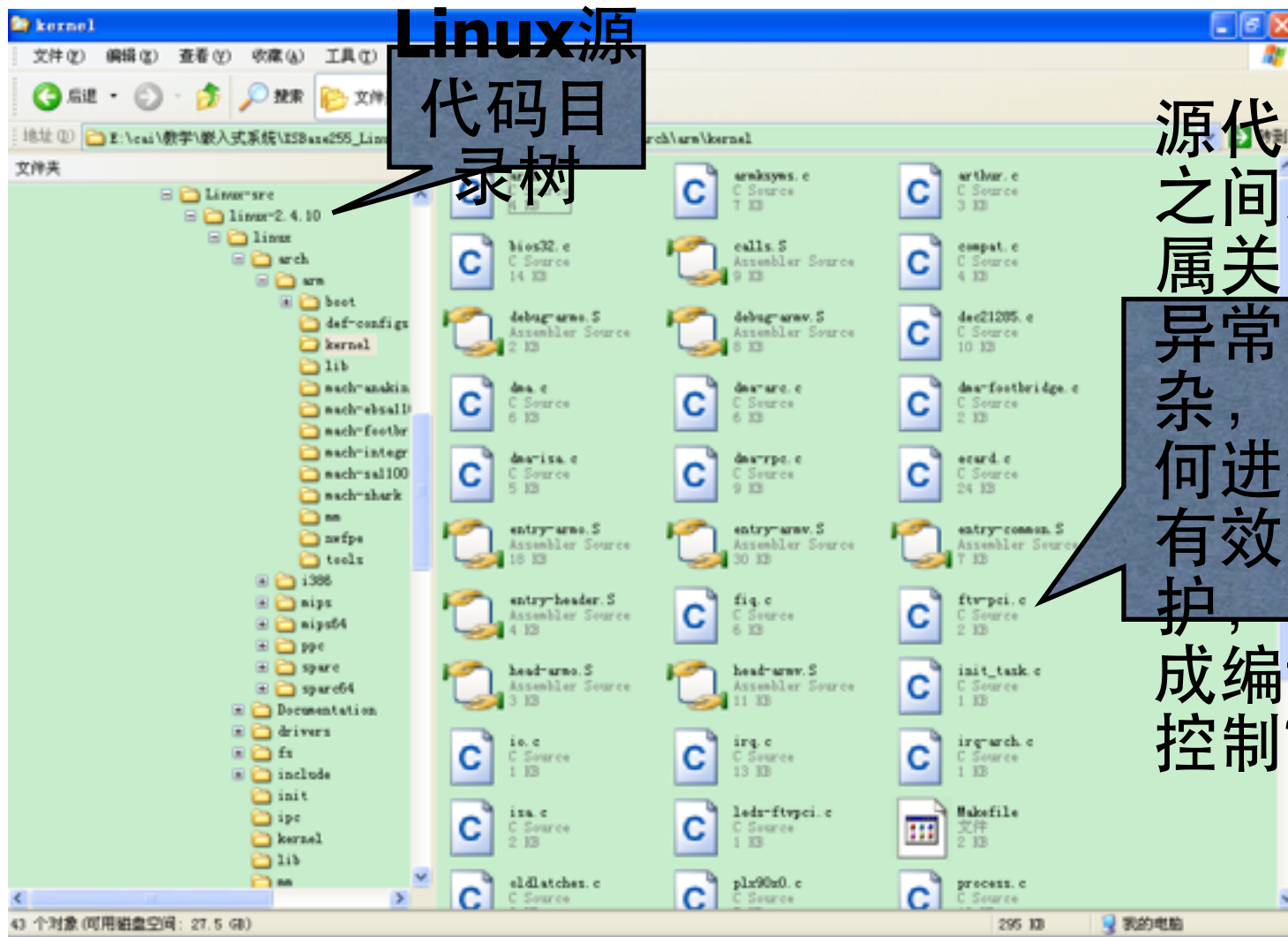
绝对地址

```
08048328 <main>:
8048328: 55          push    %ebp
8048329: 89 e5      mov     %esp,%ebp
804832b: 57          push    %edi
804832c: 56          push    %esi
804832d: 83 ec 10   sub     $0x10,%esp
8048330: 83 e4 f0   and     $0xffffffff0,%esp
8048333: b8 00 00 00 00 mov     $0x0,%eax
8048338: 29 c4      sub     %eax,%esp
804833a: 8d 7d e8   lea     0xffffffffe8(%ebp),%edi
804833d: be 0c 84 04 08 mov     $0x804840c,%esi
8048342: fc          cld
8048343: b9 0e 00 00 00 mov     $0xe,%ecx
8048348: f3 a4      repz    nouseb %ds:(%esi),%es:(%edi)
804834a: 83 ec 0c   sub     $0xc,%esp
804834d: 8d 45 e8   lea     0xffffffffe8(%ebp),%eax
8048350: 50          push    %eax
8048351: e8 12 ff ff ff call    8048268 <_init+0x38>
8048356: 83 c4 10   add     $0x10,%esp
8048359: 8d 65 f8   lea     0xfffffffff8(%ebp),%esp
804835c: 5e          pop     %esi
804835d: 5f          pop     %edi
804835e: c9          leave   %eax
804835f: c3          ret
```

重定位指令

`a.out`

Make概述



GNU Make概述

- ◆ make是一种大型软件的开发中常常使用的编译控制工具。通过指定要编译的文件和它所依赖的文件，make使用时间戳来决定是否要重新更新目标文件。
- ◆ make工具依赖一个特殊的makefile文本文件，这个文件描述了系统中各个模块之间的依赖关系。
- ◆ makefile默认文件名为makefile或Makefile。
- ◆ make系统提供了众多的函数、宏定义、隐含规则，结合shell的脚本，构成一套强大灵活的编译系统。

Make规则介绍

- ◆ 一个简单的Makefile描述规则组成：

TARGET... : PREREQUISITES...
COMMAND

...

...

- ◆ **target:** 规则的目标。通常是程序中间或者最后需要生成的文件名。可以是.o文件、也可以是最后的可执行程序的文件名。另外，目标也可以是一个make执行的动作名称，如目标“clean”，称这样的目标为“伪目标”。
- ◆ **prerequisites:** 规则的依赖。生成规则目标所需要的文件名列表。通常一个目标依赖于一个或者多个文件。依赖条件可省略。
- ◆ **command:** 规则的命令行。是make程序所有执行的动作（任意的shell命令或者可在shell下执行的程序）。命令行可省略。

Makefile举例

```
#sample Makefile
edit : main.o kbd.o command.o display.o \
      insert.o search.o files.o utils.o
      cc -o edit main.o kbd.o command.o display.o \
      insert.o search.o files.o utils.o
main.o : main.c defs.h
      cc -c main.c

kbd.o : kbd.c defs.h command.h
      cc -c kbd.c
command.o : command.c defs.h command.h
      cc -c command.c
display.o : display.c defs.h buffer.h
      cc -c display.c
insert.o : insert.c defs.h buffer.h
      cc -c insert.c
search.o : search.c defs.h buffer.h
      cc -c search.c
files.o : files.c defs.h buffer.h command.h
      cc -c files.c
utils.o : utils.c defs.h
      cc -c utils.c
clean :
      rm edit main.o kbd.o command.o display.o \
      insert.o search.o files.o utils.o
```



使用举例

- make
- make main.o
- make clean
- make -f filename

指定变量

- ◆ “objects”作为一个变量，它代表所有的.o文件的列表。在定义了此变量后，我们就可以在需要使用这些.o文件列表的地方使用“\$(objects)”来表示它，而不需要罗列所有的.o文件列表。因此上例的规则就可以这样写：

```
objects = main.o kbd.o command.o display.o \  
    insert.o search.o files.o utils.o  
edit : $(objects)  
    cc -o edit $(objects)  
.....  
.....  
clean :  
    rm edit $(objects)
```

隐含规则

- ◆ 使用**make**内嵌的隐含规则，我们的**Makefile**中就不需要明确给出重建某一个目标的命令，甚至可以不用写出明确的规则。**make**会自动根据已存在（或者可以被创建）的源文件类型来启动相应的隐含规则。例如：

foo : foo.o bar.o

cc -o foo foo.o bar.o \$(CFLAGS) \$(LDFLAGS)

- ◆ GNU make的内嵌隐含规则：

1. 编译C程序

“*.o”自动由“*.c”生成，执行命令为“\$(CC) -c \$(CPPFLAGS) \$(CFLAGS)”。

2. 编译C++程序

“*.o”自动由“*.cc”或者“*.C”生成，执行命令为“\$(CXX) -c \$(CPPFLAGS) \$(CFLAGS)”。

模式规则

- ◆ 模式规则类似于普通规则，只是在模式规则中，目标的定义中需要包含“%”字符（确切地说是一个），包含“%”的目标被用来匹配一个文件名，“%”可以匹配任何非空字符串。规则的依赖文件中同样可以使用“%”，依赖中的“%”的取值情况由目标中的“%”来决定。例如：模式规则“%.o : %.c”表示了所有的.o文件是由.c文件来生成的。我们可以使用模式规则来定义一个隐含规则。

- ◆ 举例：

%.o : %.c

\$(CC) -c \$(CFLAGS) \$(CPPFLAGS) \$< -o \$@

- ◆ 老式风格定义隐含规则的方式：

.c.o:

\$(CC) -c \$(CFLAGS) \$(CPPFLAGS) \$< -o \$@

自动变量

◆ \$@

代表规则中的目标文件名。如果目标是一个文档（Linux中，一般称.a文件为文档），那么它代表这个文档的文件名。在多目标的模式规则中，它代表的是那个触发规则被执行的目标文件名。

◆ \$<

规则的第一个依赖文件名。如果是隐含规则，则它代表通过目标指定的第一个依赖文件。

◆ 举例：

main.o: main.c

\$(CC) -c \$(CFLAGS) \$(CPPFLAGS) \$< -o \$@

其中：**\$< = main.c** **\$@ = main.o**

关于生成交叉编译器 (1/3)

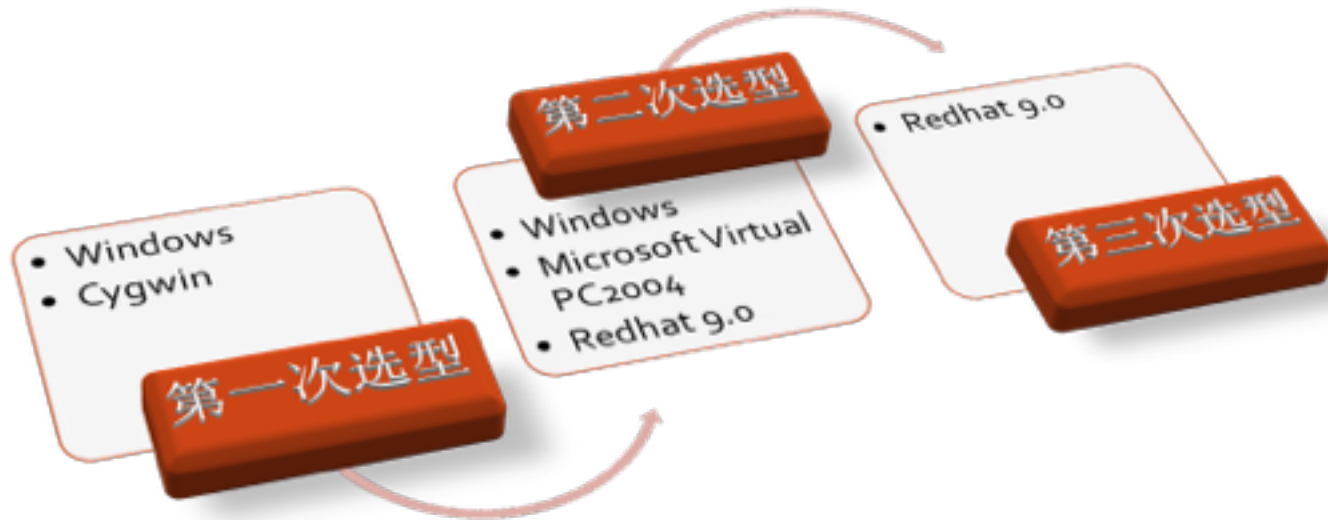
◆ 为什么需要生成交叉编译器?

针对特定的嵌入式体系结构，并一定有现成的交叉编译器，因而，我们不得不使用现有的GCC代码来生成交叉编译器！

关于生成交叉编译器 (2/3)

◆ 宿主机平台如何进行选择?

- 模拟环境
- 真机环境



SPARC V8交叉编译宿主平台选型实例

关于生成交叉编译器

(3/3)

◆ 交叉编译器生成的过程如何?

1. 制作交叉的binutils二进制工具
2. 制作不带库的gcc交叉编译器
3. 用制作好的gcc交叉编译器，生成所需要的C库（glibc、newlib、uclibc等）
4. 重新编译带库的gcc，生成完整的交叉编译器