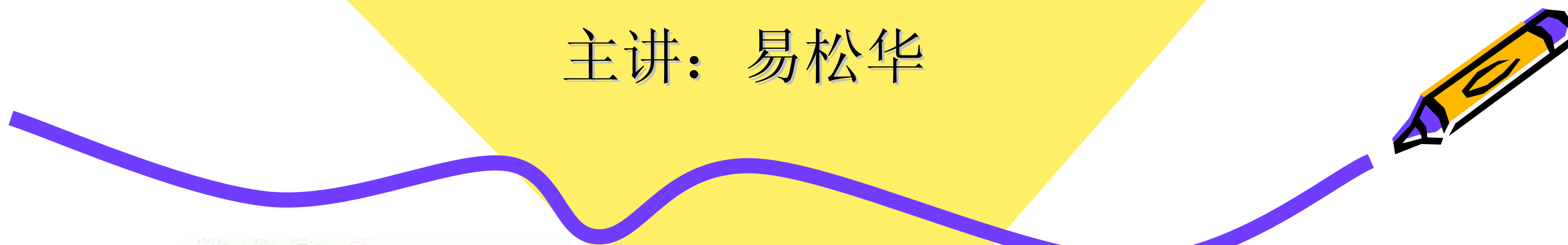
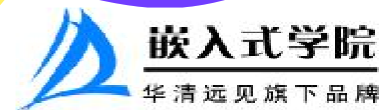
嵌入式Linux调试技术

如何在嵌入式LINUX中调试

主讲：易松华





- 调试简介
- Linux的编译调试工具、仿真器介绍
- 应用程序的调试
- 启动代码调试
- 内核调试
- 其他调试方法

说在前面的话

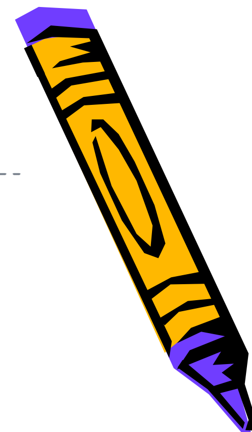
- 八大菜系
  – 各具特色
- 调试技术
  – 博大精深

理想：

- 1. 吃遍天下美食
    - 蔡澜提菜篮
- 2. 成为“无米之厨”
- 3.成为神仙
    - 吃喝之事完全变成“打酱油”
- 现实：

限于条件，最爱只能麻婆豆腐

限于个人经验，只能涉及LINUX的部分调试技术



- 调试(Debug) = De +bug
  - 指发现和去除软件失效根源的过程。

- 软件调试的简单分类

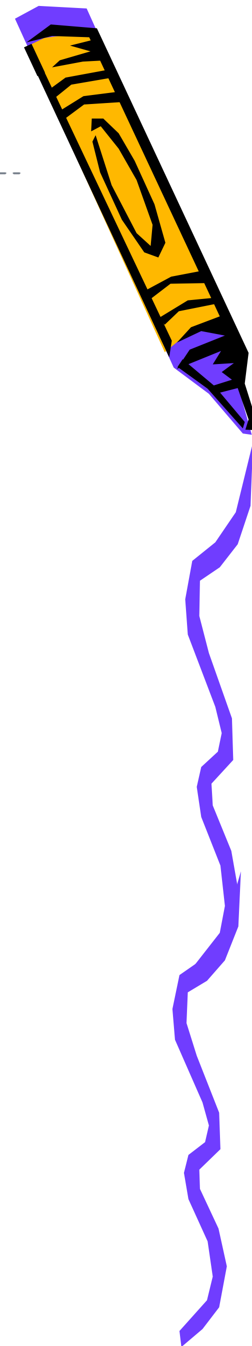
分类方法	类别1	类别2
调试所处阶段	静态调试	动态调试
代码类别	机器级调试	源代码调试
调试对象所处位置	应用调试	系统级调试
运行环境	本地调试	交叉（远程）调试

调试简介

- 软件调试的关键技术----断点
  - 软件断点
    - CPU在自陷或异常时能跳转到确定位置执行代码
    - 靠程序指令陷阱来实现,即通过**修改程序指令**的方法来实现
  - 硬件断点
    - 硬件断点需要目标CPU的硬件支持
  - 关系
    - 断点个数
      - 软件断点不限
      - 硬件断点受限于CPU的设置
        - » Arm7/arm9 2个,ARM11 8个
    - 应用场合
      - 软件断点主要用于RAM,
        - » 如有比较高级的仿真器关持，可支持NOR flash
      - 硬件断点可设置在任何位置代码上

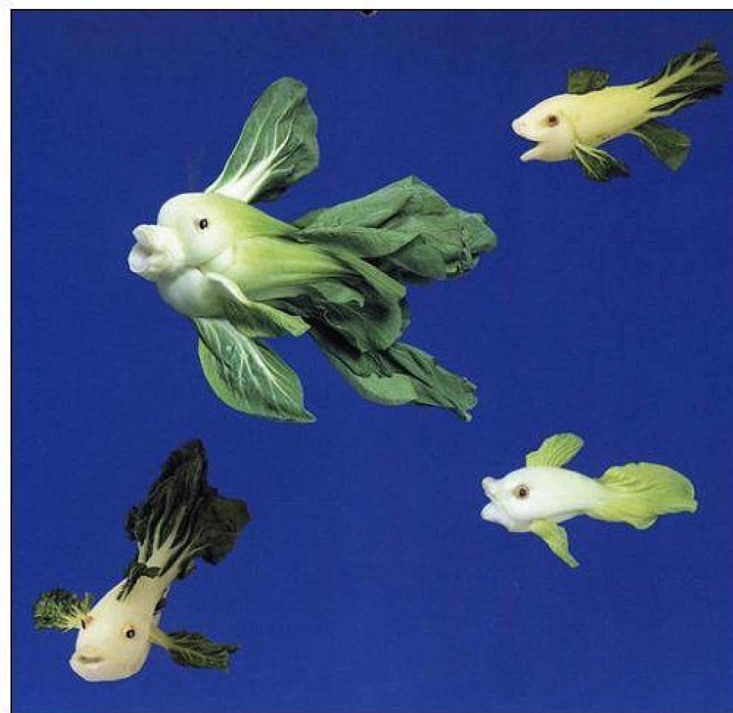
调试简介

- 调试接口
  - 调试的起源--ICE（In Circuit Emulation）
  - Motorola用于M68K和PowerPC的调试接口—BDM
  - 流行风向标--JTAG
  - “软件厨师”的梦中情人--软件仿真



调试简介

- 调试应遵循的规则
  - 调试器自身须稳定，须反映真实的信息
  - 提供尽可能多反映真实信息
  - 尽可能减少对被测试系统的影响
    - 软件调试时一种有损测试



Linux的编译调试工具、仿真器介绍

- Binutils
  - objdump反汇编，查看目标文件或可执行文件内部信息。
  - addr2line把机器地址转换到代码对应的位置。
  - nm查看目标文件或可执行文件中的各种符号。
  - gprof分析各个函数的使用情况，找出性能的瓶颈所在(这需要加编译选项)。
- Gdb/KDB/KGDB
- 内核
  - OOPS和PANIC

Linux的编译调试工具、仿真器介绍

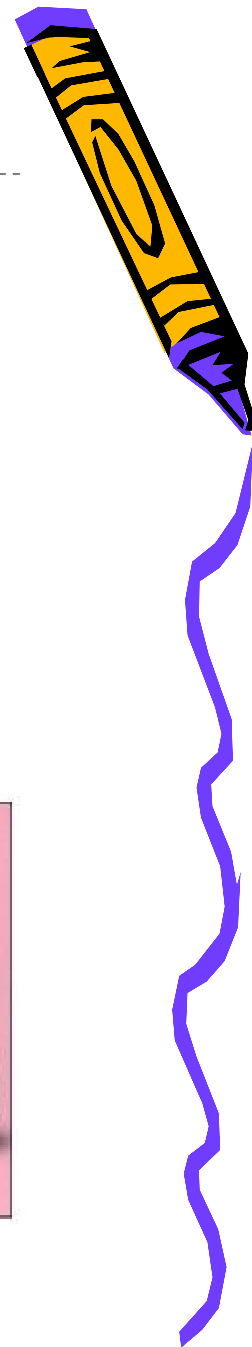
百问网OpenOCD仿真器

Abatron BDI3000

Lauterbach-Trace32

应用程序的调试

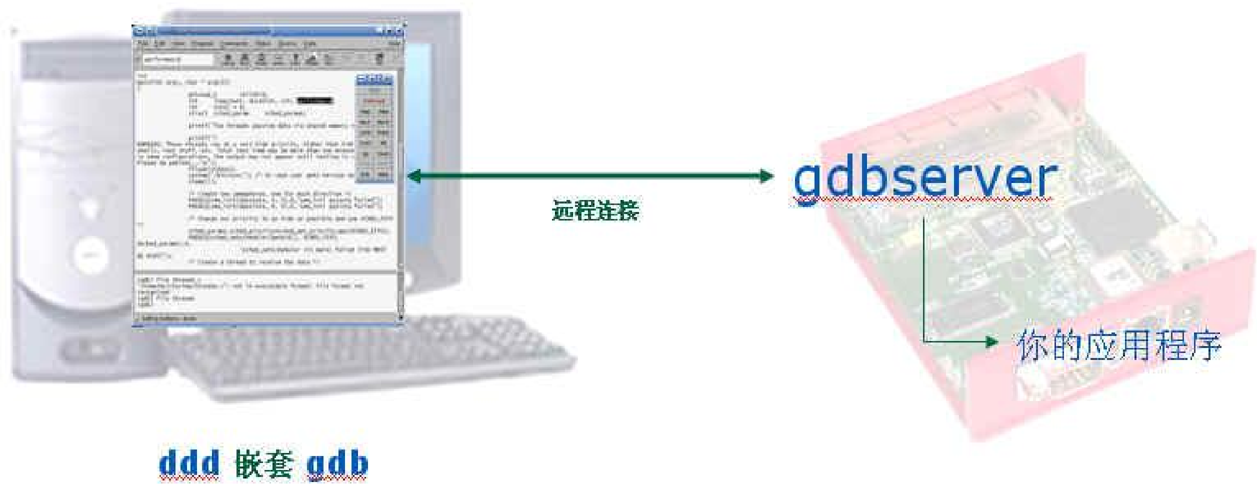
- 本地和交叉调试
- 多进程调试
- 多线程调试

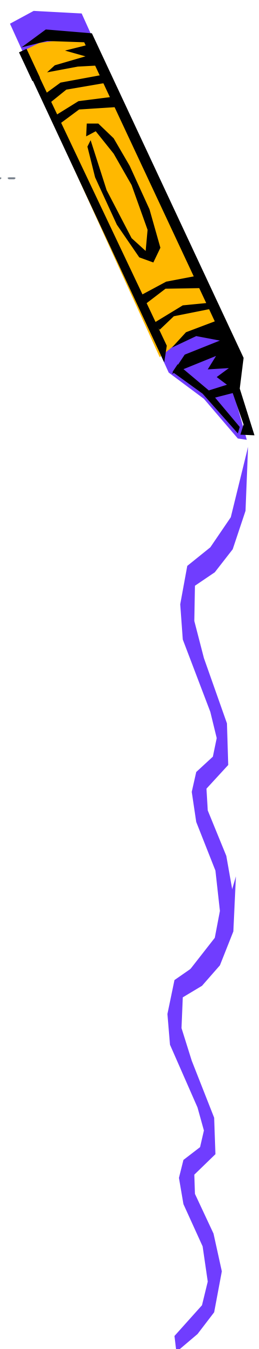


应用程序的调试

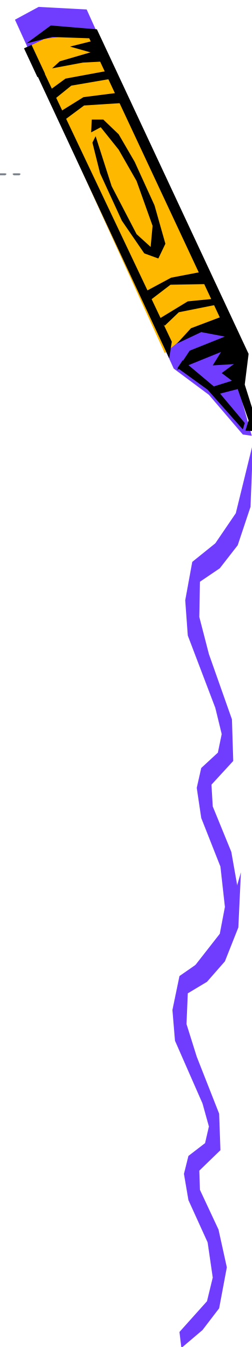
嵌入式学院 华清远见旗下品牌

- 本地和交叉调试
  - gdb/ddd/eclipse
  - gdbserver/ddd



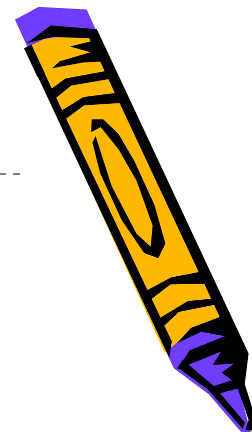


嵌入式学院
华清远见旗下品牌

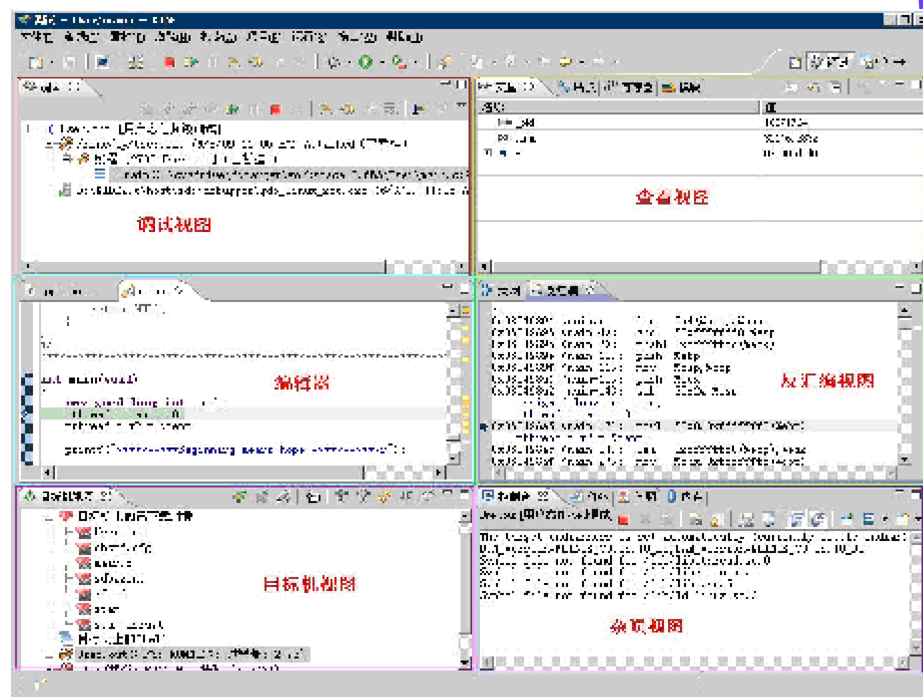


- 多进程调试
  - follow-fork-mode
  - attach 子进程
  - GDB wrapper 方法

应用程序的调试

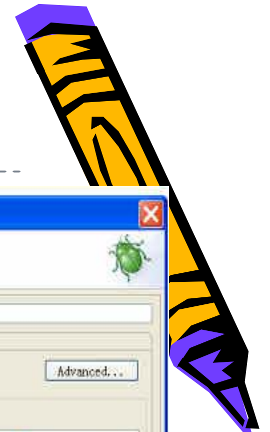


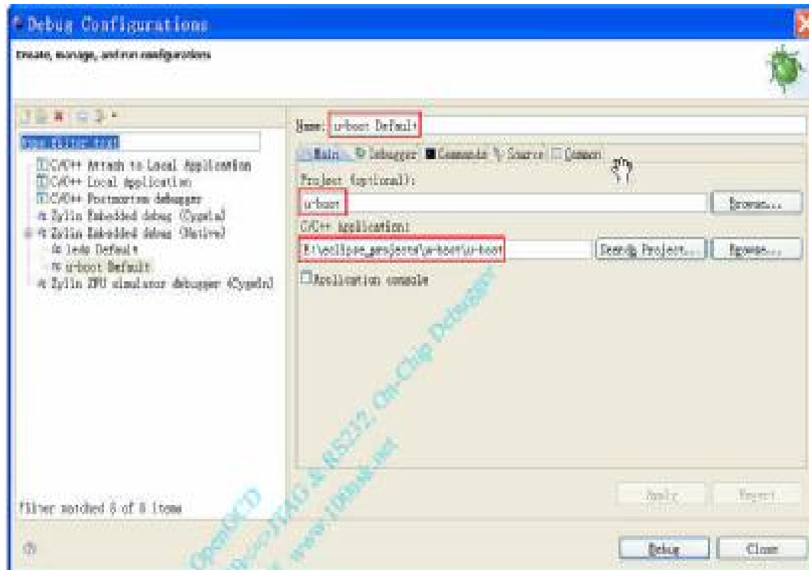
- 多线程调试（gdb）
  - info threads
  - thread ID
  - break thread_func
  - thread apply ID1 ID2 command
  - Etc.

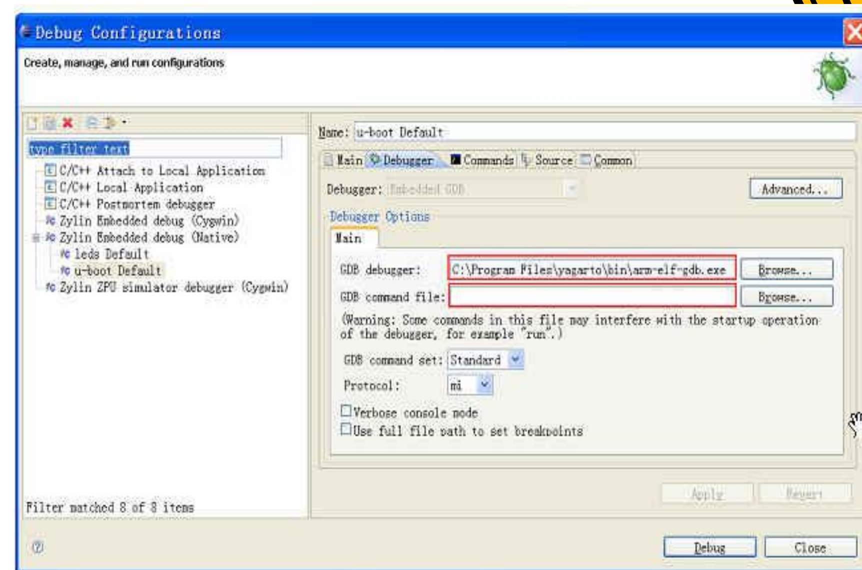


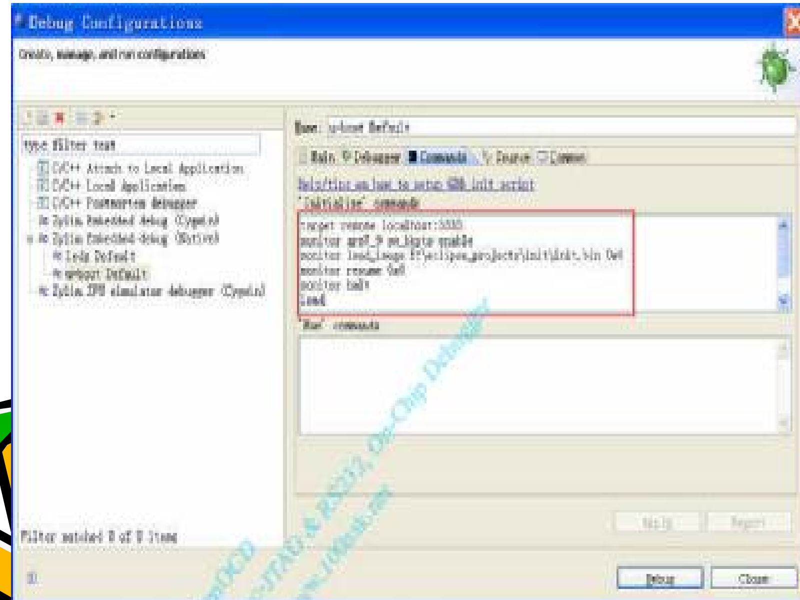
启动代码调试-openOCD调试uboot

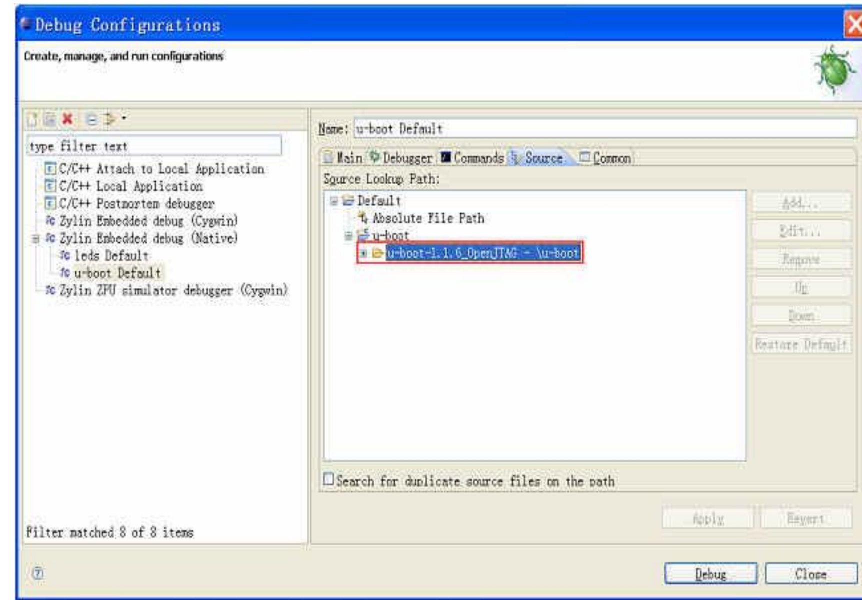
嵌入式学院
华清远见旗下品牌





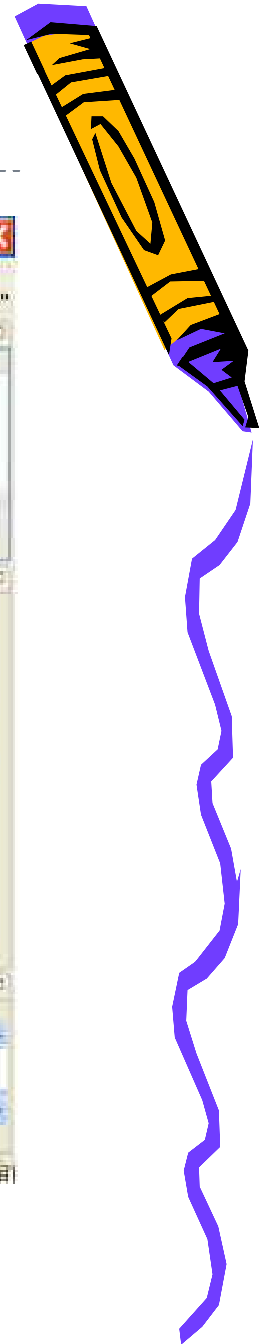






启动代码调试-openOCD调试uboot

嵌入式学院
华清远见旗下品牌



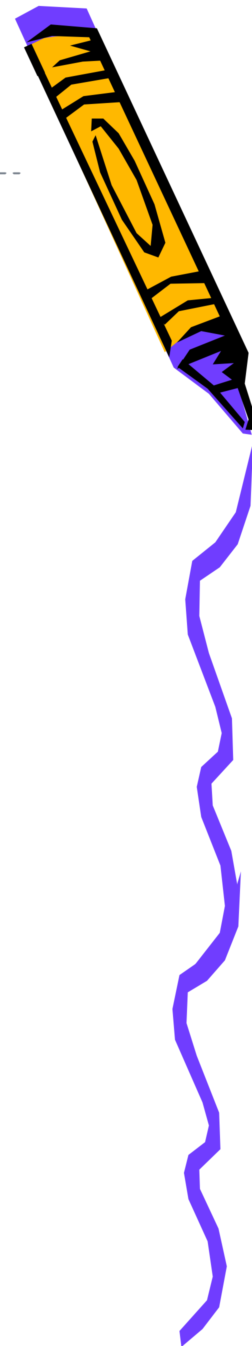
```
Debug - u-boot/u-boot-1.1.6_OpenJTAG/lib_arm/board.c - Eclipse Platform
File Edit Refactor Navigate Search Run Project Window Help

Debug
  OpenOCD [Program]
    C:\Program Files\openocd-r717\bin\openocd-ftd2xx.exe
  u-boot Default [Zylin Embedded debug (Native)]
    Embedded GDB (08-9-24 上午3:58) (Suspended)
      Thread [0] (Suspended)
        2 start_armboot() board.c:248 0x33f81558
        1 stack_setup() start.S:153 0x33f800c0
    C:\Program Files\yagarto\bin\arm-elf-gdb.exe (08-9-24 上午3:58)

Variables  Breakpoints  Registers  Modules
Name            Value
init_fnc_ptr    0x0000000a
s               0x33f80000
size            10

board.c
#if defined(CONFIG_VFD) || defined(CONFIG_LCD)
    unsigned long addr;
#endif

    /* Pointer is writable since we allocated a register for it */
    gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
    /* compiler optimization barrier needed for GCC >= 3.4 */
    __asm__ __volatile__("": : :"memory");

    memset ((void*)gd, 0, sizeof (gd_t));
    gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
    memset (gd->bd, 0, sizeof (bd_t));

    monitor_flash_len = _bss_start - _armboot_start;

    for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
        if ((*init_fnc_ptr)() != 0) {
            hang ();

Console  Tasks  Problems  Executables  Memory
u-boot Default [Zylin Embedded debug (Native)] C:\Program Files\yagarto\bin\arm-elf-gdb.exe (08-9-24 上午3:58)
Breakpoint 1, start_armboot () at board.c:248
    in board.c
248     board.c: No such file or directory.
No source file named leds.c.
```

OpenOCD USB JTAG & RS232, On-Chip Debugger
百问网 www.100ask.net

内核代码调试

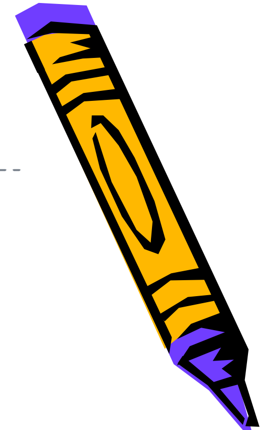
- 内核解压缩和MMU使能之前代码调试
  - 类似UBOOT
  - 板级初始化
    - 通过仿真器初始化
    - 通过bootloader初始化
    - 调试的是Image

内核生成步骤

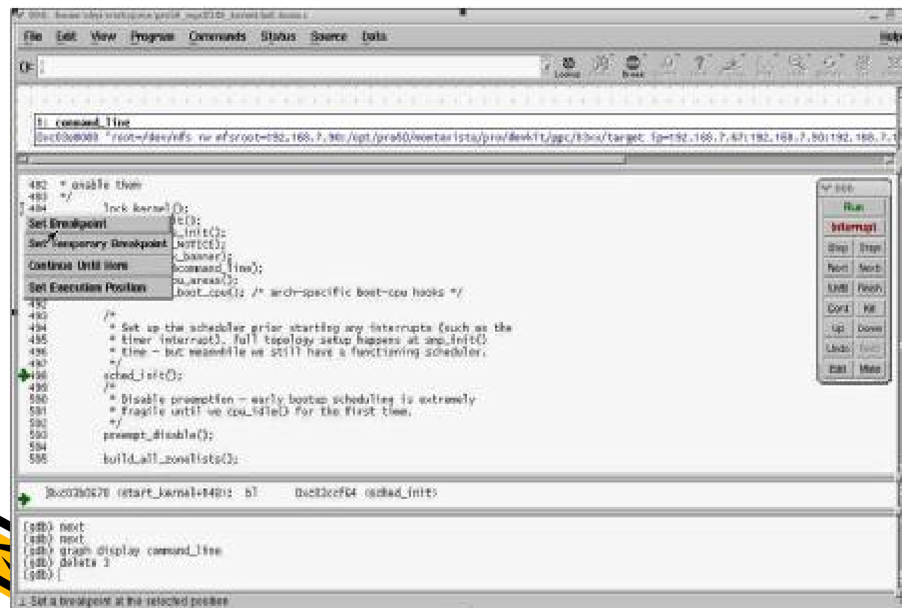


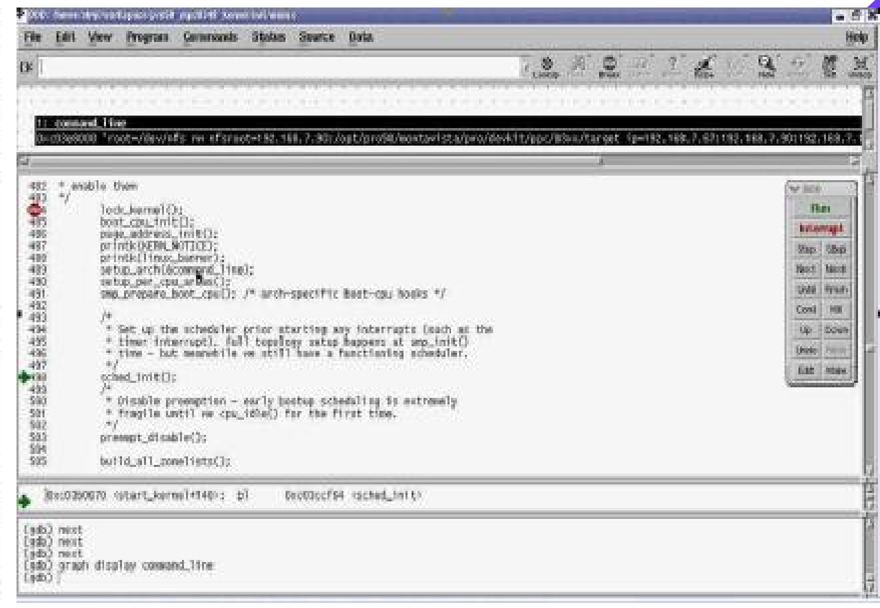
嵌入式学院
华清远见旗下品牌

内核代码调试

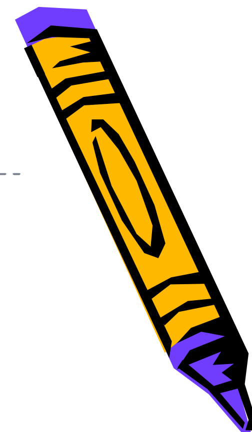


- 内核MMU使能之后的代码调试
  - 内核修改和配置
  - 配置内核
  - 调试内核





内核代码调试

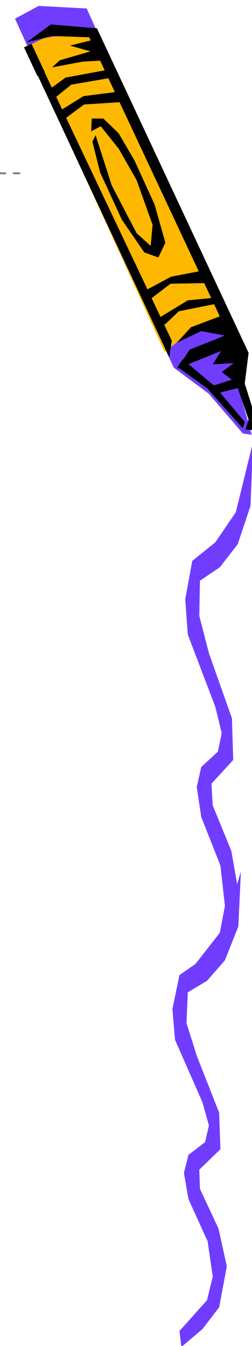


- 内核驱动调试
  - 装载驱动和符号表
  - 设置断点
  - 调试

```
DDD: /opt/pro50/montavista/pro/devkit/ppc/83xx/target/root/examples/misc-modules/jit.c
File  Edit  View  Program  Commands  Status  Source  Data                Help

3: start
(char **) 0xcfbb7e9c

4: *start
0xc03cc000 "0xffff1414 0x00000000ffff1414 946693509.452254\n"

101
102         /* get them four */
103         j1 = jiffies;
104         j2 = get_jiffies_64();
105         do_gettimeofday(&tv1);
106         tv2 = current_kernel_time();
107
108         /* print */
109         len=0;
110         len += sprintf(buf,"0x%08lx 0x%016Lx %10i.%06i\n"
111                        "%40i.%09i\n",
112                        j1, j2,
113                        (int) tv1.tv_sec, (int) tv1.tv_usec,
114                        (int) tv2.tv_sec, (int) tv2.tv_nsec);
115         *start = buf;
116         return len;
117 }
118
119 /*
120  * The timer example follows
121  */
122
123 int tdelay = 10;
124 module_param(tdelay, int, 0);

0xd106e078 <jit_currentime+120>:       lwz     r11,0(r1)

(gdb) next
(gdb) next
(gdb) next
' ' <repeats 31 times>, "946693509.448000005\n"(gdb) graph display *start
(gdb)
Display 4: *start (enabled, scope jit_currentime, address 0xcfbb7e9c)
```

其他调试方法

- 进程跟踪工具 strace
- 系统性能测试 gprof
- 代码覆盖率测试 gcov
- core dump
- 内存泄漏检测工具



用strace 跟踪程序的系统调用和信号

嵌入式学院 华清远见旗下品牌

- 功能
  - 单个linux进程的跟踪工具，能跟踪并打印出程序调用的所有系统调用和信号
  - 不需要重新编译被跟踪的程序
  - 开源工具 http://sourceforge.net/projects/strace/
- 语法
  - strace [-dffhiqrtttTvxx] [-acolumn] [-eexpr] ... [-ofile] [-ppid] ... [-sstrsize] [-uusername] [command [arg ...]]
    strace -c [-eexpr] ... [-Ooverhead] [-Ssortby] [command [arg ...]]

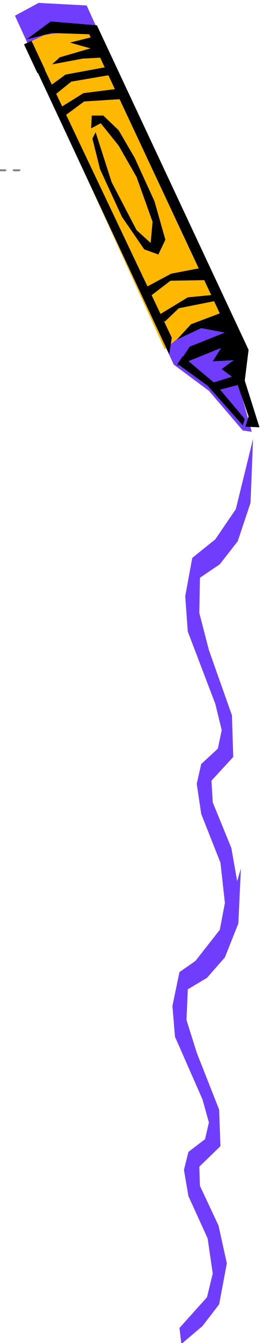
- 例子
  - strace –f –o ls.starace ls

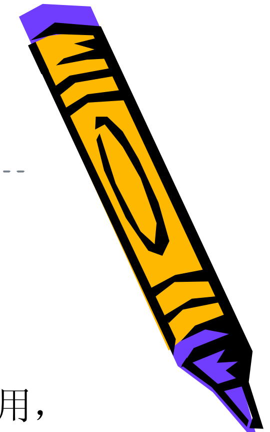
交叉编译 CC=arm-linux-gcc ./configure –host=arm-linux (参看其文档）

用 gprof 测试程序执行的时间

嵌入式学院
华清远见旗下品牌

- gprof 是代码执行测试工具，它可以测试程序中各个函数执行所花的时间
- 可以用gprof来优化代码
- 编译测试代码时要指定 –pg 选项，不能使用-O2选项
- 包含在 gnu binutils 工具集中 http://www.gnu.org/software/binutils/
- 语法
  - Usage: gprof [-[abcDhilLsTvwxyz]] [-[ACeEfFJnNOpPqQZ][name]] [-I dirs]
  - [-d[num]] [-k from/to] [-m min-count] [-t table-length]
  - [--[no-]annotated-source[=name]] [--[no-]exec-counts[=name]]
  - [--[no-]flat-profile[=name]] [--[no-]graph[=name]]
  - [--[no-]time=name] [--all-lines] [--brief] [--debug[=level]]
  - [--function-ordering] [--file-ordering]
  - [--directory-path=dirs] [--display-unused-functions]
  - [--file-format=name] [--file-info] [--help] [--line] [--min-count=n]
  - [--no-static] [--print-path] [--separate-files]
  - [--static-call-graph] [--sum] [--table-length=len] [--traditional]
  - [--version] [--width=n] [--ignore-non-functions]
  - [--demangle[=STYLE]] [--no-demangle] [@FILE]
  - [image-file] [profile-file...]
  - Report bugs to <URL:http://www.sourceware.org/bugzilla/>





使用 gcov 测试代码覆盖率

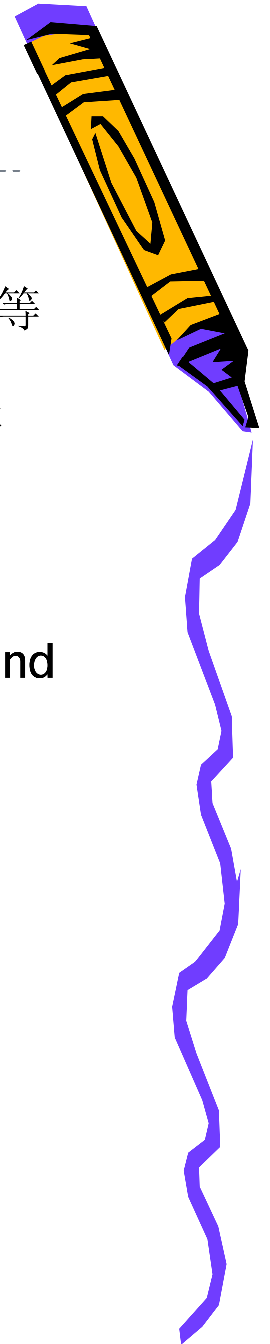
- gcov是代码覆盖率测试工具，它可以分析程序源代码行的调用次数，看其中哪些频繁调用，哪些没有调用过
- 可以用来优化和测试程序（分支遍历测试）
- 包含在 gnu binutils 工具集中 http://www.gnu.org/software/binutils
- 语法
  - Usage: gcov [OPTION]... SOURCEFILE
  - Print code coverage information.
  - -h, --help Print this help, then exit
  - -v, --version Print version number, then exit
  - -a, --all-blocks Show information for every basic block
  - -b, --branch-probabilities Include branch probabilities in output
  - -c, --branch-counts Given counts of branches taken rather than percentages
  - -n, --no-output Do not create an output file
  - -l, --long-file-names Use long output file names for included source files
  - -f, --function-summaries Output summaries for each function
  - -o, --object-directory DIR|FILE Search for object files in DIR or called FILE
  - -p, --preserve-paths Preserve all pathname components
  - -u, --unconditional-branches Show unconditional branch counts too
  - For bug reporting instructions, please see: <URL:http://gcc.gnu.org/bugs.html>.
  - For Debian GNU/Linux specific bug reporting instructions, please see:
  - <URL:file:///usr/share/doc/gcc-4.1/README.Bugs>

core dump

- 当程序运行过程中发生异常, 程序异常退出时, 由操作系统把程序当前的内存状况存储在一个core文件中, 叫core dump
- 系统中默认情况下可能不产生core文件，需要用ulimit -c unlimited语句进行设置，core文件生成的位置一般在程序运行的当前目录下，文件名为core.进程号(不同的系统也许有所不同，请查看手册)
- 使用 core dump
  - 编译程序是指定-g选项
  - 运行程序，发生异常，产生 core 文件
  - gdb test core.XXX
  - 之后可以用gdb的where命令查看

内存泄漏检测工具-valgrind

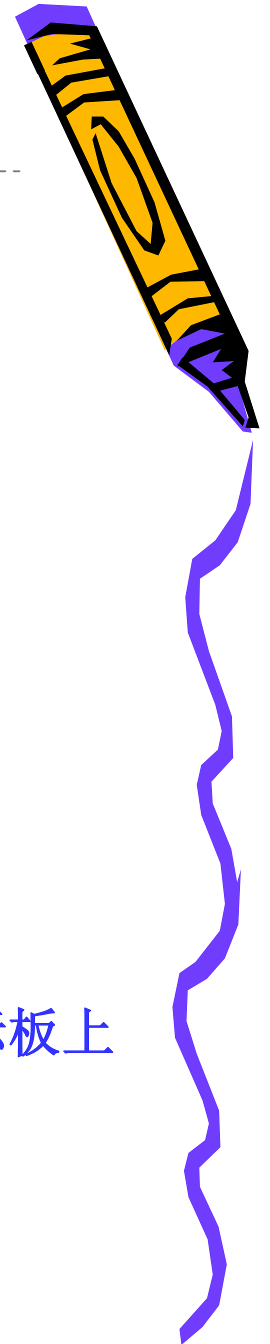
嵌入式学院
华清远见旗下品牌



- valgrind
  – Linux下检测应用程序的内存管理问题：泄漏，越界、未初始化等等
  – 可以检查程序运行时的内存泄漏问题
  – 不需要和被测试软件一起编译，但要用valgrind加载被调试的程序
  – 开源 http://www.valgrind.org/
- valgrind安装
  – 可以由源码安装
  – 安装编译好的二进制文件，如：ubuntu下用apt-get install valgfind 等
- valgrind简单用法
  – 编译被检测的代码指定 –g 选项
  – 用valgrind 加载编译好的程序
  – 操作，直至程序退出
  – valgrind给出内存情况报告
- valgrind目前支持
  – X86/amd64/ppc32/ppc64/
  – linux

内存泄漏检测工具-memwatch

嵌入式学院
华清远见旗下品牌

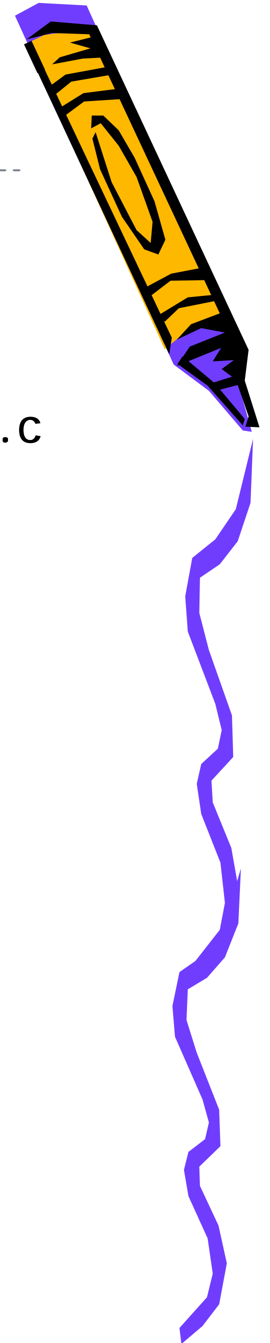


- memwatch 是另一个c语言内存错误检测工具
- 开源软件，由 Johan Lindh 编写，可以在这里 http://www.linkdata.se/sourcecode.html下载
- memwatch能检测到
  - 没有释放的内存（unfreed memory）
  - 两次释放（double-free）
  - 错误释放（erroneous free）
  - 越界
- memwatch的使用
  - 包含memwatch.h头文件
  - 编译指定"-DMEMWATCH"和 "-DMW_STDIO"（MEMWATCH_STDIO）选项
  - 和应用程序一起编译memwatch.c
  - 运行编译好的程序，产生日志

memwatch可以运行在开发主机上，也可以交叉编译后在目标板上运行

其他见memwatch文档和代码

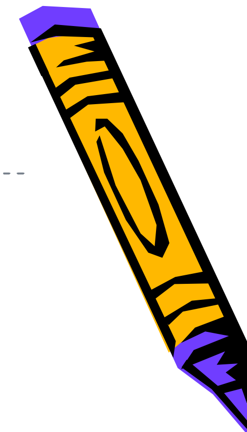
测试 memwatch



- 下载 memwatch
- 编译 test.c
  - gcc -DMEMWATCH -DMEMWATCH_STDIO -o test test.c memwatch.c
- 运行编译完的程序
  - ./test
  - 运行结束将产生memwatch.log文件
    - 内存使用的全局信息统计，包括四点：
      - 1）分配了多少次内存
      - 2）最大内存使用量
      - 3）分配的内存总量
      - 4）为释放的内存总数
    - message: <sequence-number> filename(linenumber), information

memwatch.log 样例

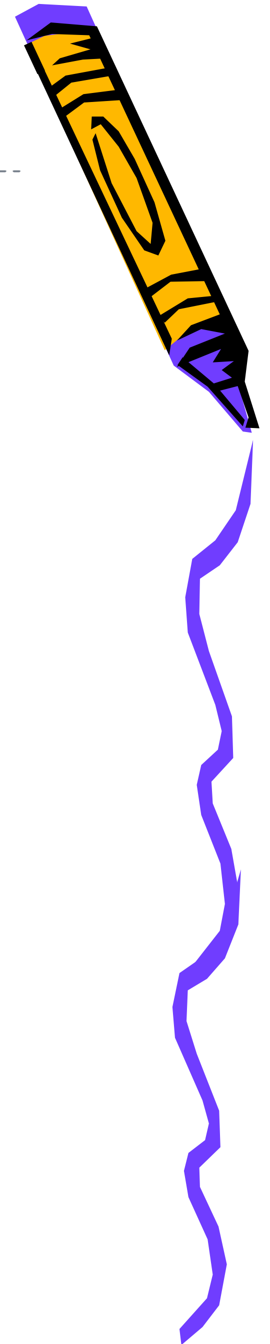
嵌入式学院
华清远见旗下品牌



- ============= MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh =============
- Started at Thu Oct 9 14:56:39 2008
- Modes: __STDC__ 32-bit mwDWORD==(unsigned long)
- mwROUNDALLOC==4 sizeof(mwData)==32 mwDataSize==32
- statistics: now collecting on a line basis
- Hello world!
- underflow: <5> test.c(62), 200 bytes alloc'd at <4> test.c(60)
- relink: <7> test.c(66) attempting to repair MW-0x8051390...
- relink: MW-0x8051390 is the head (first) allocation
- relink: successful, no allocations lost
- limit: old limit = none, new limit = 1000000 bytes
- grabbed: all allowed memory to no-mans-land (976 kb)
- Killing byte at 0x805d704
- Killing byte at 0x80588e4
- Killing byte at 0x805179c
- Killing byte at 0x8051350
- check: <7> test.c(95), checking chain alloc nomansland
- check: <7> test.c(95), complete; no errors
- internal: <9> test.c(105), checksum for MW-0x8147210 is incorrect
- overflow: <9> test.c(105), 0 bytes alloc'd at <8> test.c(131169)
- internal: <9> test.c(107), no-mans-land MW-0x8147210 is corrupted
- realloc: <9> test.c(107), 0x8147234 was freed from test.c(105)
- WILD free: <10> test.c(110), unknown pointer 0x80489e4

memwatch.log

memwatch.log（续）



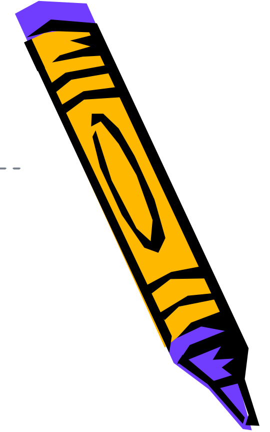
- Stopped at Thu Oct 9 14:56:39 2008

- wild pointer: <10> no-mans-land memory hit at 0x805d704
- wild pointer: <10> no-mans-land memory hit at 0x80588e4
- dropped: all no-mans-land memory (975 kb)
- internal: the grab list is not empty after mwDropAll()
- unfreed: <3> test.c(59), 20 bytes at 0x80511dc {FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE FE ....}

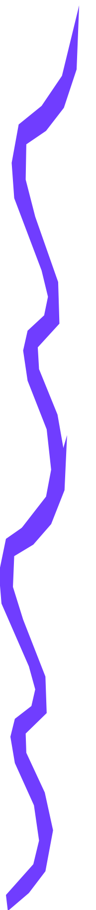
- Memory usage statistics (global):
- N)umber of allocations made: 5
- L)argest memory usage : 12020
- T)otal of all alloc() calls: 12530
- U)nfreed bytes totals : 12020

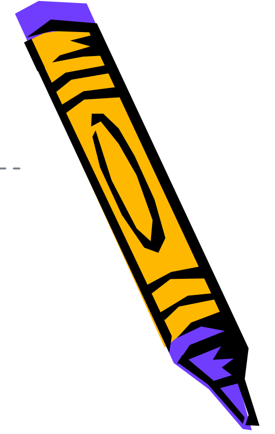
- Memory usage statistics (detailed):

Module/Line	Number	Largest	Total	Unfreed
test.c	5	12020	12530	12020
131169	0	0	0	0
97	1	12000	12000	12000
64	1	100	100	0
60	1	200	200	0
59	1	20	20	20
57	1	210	210	0



Q&A





谢谢！