CIS 450 – Computer Architecture and Organization

Lecture 26: Profiling

Mitch Neilsen
(neilsen@ksu.edu)
219D Nichols Hall

Topics

- **Debugging and Profiling**
  - **Available Tools**
- **BrickOS RTOS**

GDB (GNU DeBugger)

Basic functionality:

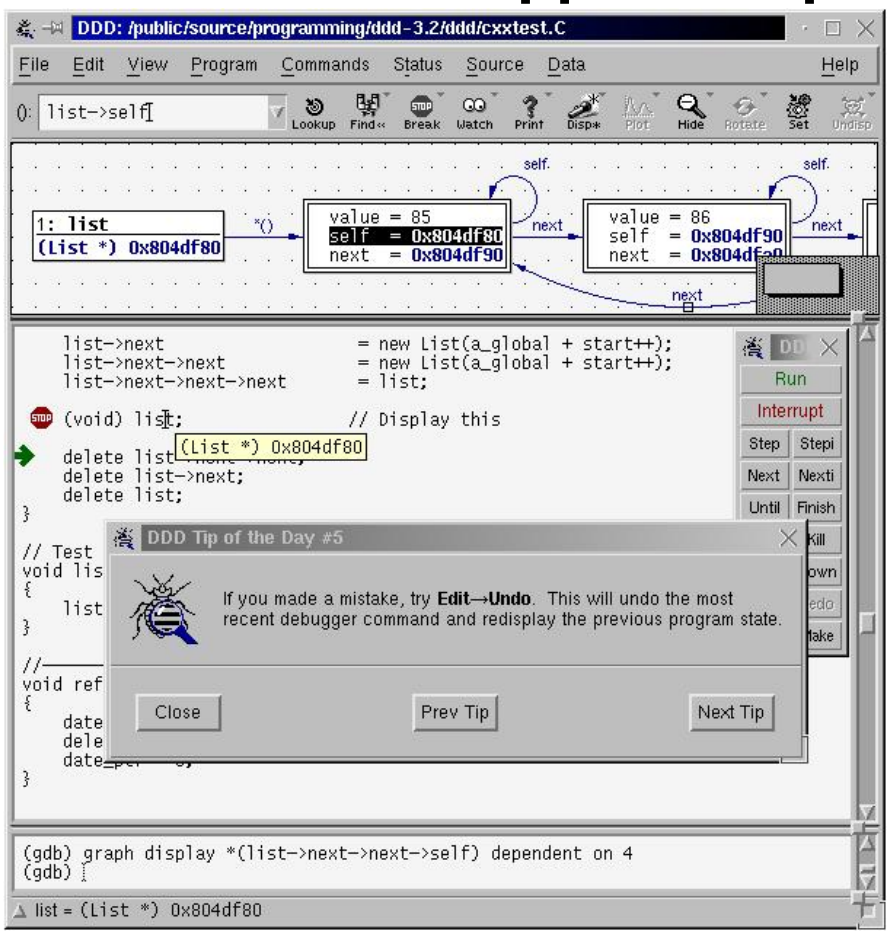
- **Can run programs in an observable environment**
- **Uses `ptrace`-interface to insert breakpoint, single step, inspect, and change registers and variables**
- **Does not require compilation with “-g”, but works much better if it has the symbol tables available**
- **Maintains source line numbers and can inspect source files**

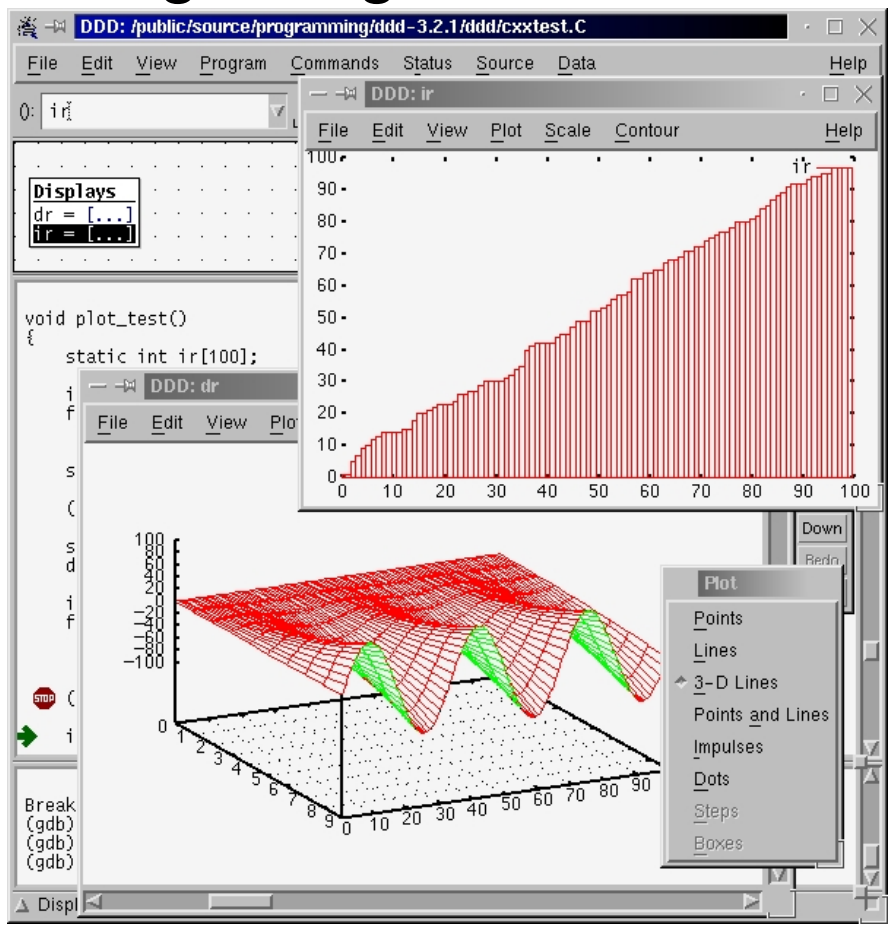
- **Ability to attach to a running process**
- **Ability to watch memory locations**
- **Conditional breakpoints**

- **Some graphical user interfaces exist (DDD, KDbg, …)**

DDD

Graphical front-end to GDB with extended data visualization support: http://www.gnu.org/software/ddd





Annoyingly Frequent Case

Memory corruption due to an earlier pointer or dynamic memory allocation error: bug cause and effect are separated by 1000's of instructions

- **Use GDB to watch the corruption happen:**
  - **Use conditional breakpoints: `break ... if cond`**
  - **Set a watchpoint: `[r,a]watch expr`**
- **Use *dog-tags* in your program**
- **Use a debugging-version of *malloc()***
- **Use run-time verification tools**

Dogtags

GDB style watch points are frequently too slow to be used in large, complex programs.

```
#ifdef USE_DOG_TAGS
#define DOGTAG(x) int x;
#else
#define DOGTAG(x)
#endif
```

```
struct foobar {
    DOGTAG(dt1);
    int buf[20];
    DOGTAG(dt2);
};
```

- **If dogtags are enabled, maintain a list of all allocated dogtags (easier with C++ class objects using the constructor)**
- **Initialize dogtags to a distinct value (e.g. 0xdeadbeef)**
- **Provide function that checks the integrity of the dogtags**

- **When to call this function?**

Dogtags (continued)

Call check funtion near suspect codes by manually inserting calls or (*hack alert*):

```
#ifdef AUTO_WATCH_DOG_TAGS
#define if(expr) if (CHECK_WATCHED_DOG_TAGS,(expr))
#define while(expr) while (CHECK_WATCHED_DOG_TAGS,(expr))
#define switch(expr) switch (CHECK_WATCHED_DOG_TAGS,(expr))
#endif /* AUTO_WATCH_DOG_TAGS */
```

Dynamic Memory Allocation Checker

`malloc()` and friends are a frequent source of trouble therefore there are numerous debugging aids for this problem. The typical functionality include:

- **Padding the allocated area with dogtags that are checked when any dynamic memory allocation functions are called or on demand.**
- **Checking for invalid free() calls (multiple, with bad argument)**
- **Checking for access to freed memory regions**
- **Keeping statistics of the heap utilization**
- **Logging**

MALLOC_CHECK_

In recent versions of Linux libc (later than 5.4.23) and GNU libc (2.x), defining `MALLOC_CHECK_` causes extra checks to be enabled (at the expense of lower speed):

- **Checks for multiple free() calls**
- **Overruns by a single byte**

Boehm-Weiser Conservative Garbage Collector

Ref: http://www.hpl.hp.com/personal/Hans_Boehm/gc/

Idea: forget about free() calls and try to use garbage collection within C. Has to be conservative.

- **Checks for existing pointers to allocated memory regions**
- **Circular pointers prevent reclaiming**
- **Assumes that pointers point to first byte (not necessarily true)**
- **Assumes that pointers are not constructed on the fly**

Electric Fence, by Bruce Perens

Ref: http://sunsite.unc.edu/pub/Linux/devel/lang/c/ElectricFence-2.0.5.tar.gz

Idea: use the virtual memory mechanism to isolate and protect memory regions

- **Pro: very fast – uses hardware (page faults) for the testing**
- **Con: Fairly large memory overhead due to page-size granularity**

- **Variations of this idea: Wisconsin Wind-Tunnel project – uses ECC bits to get finer granularity (highly platform dependent)**

Run Time Memory Checkers

Very powerful tools that use binary translation techniques to instrument the program:

- **The program (executable or object files) is disassembled and memory access (or any other operations) are replaced with code that add extra checking**
- **Generally results in a 2-50x slow-down, depending on the level of checking desired**
- **Can be used for profiling and performance optimizations**

Valgrind (IA-32, x86 ISA)

Open source software licensed under the GPL (like Linux):
http://valgrind.org/

***Valgrind* is a general purpose binary translation infrastructure for the IA-32 instruction set architecture**

Tools based on *Valgrind* include:

- ***Memcheck* detects memory-management problems**
- ***Addrcheck* is a lightweight version of Memcheck which does no uninitialised-value checking**
- ***Cachegrind* is a cache profiler. It performs detailed simulation of the I1, D1 and L2 caches in your CPU**
- ***Helgrind* is a thread debugger which finds data races in multithreaded programs**

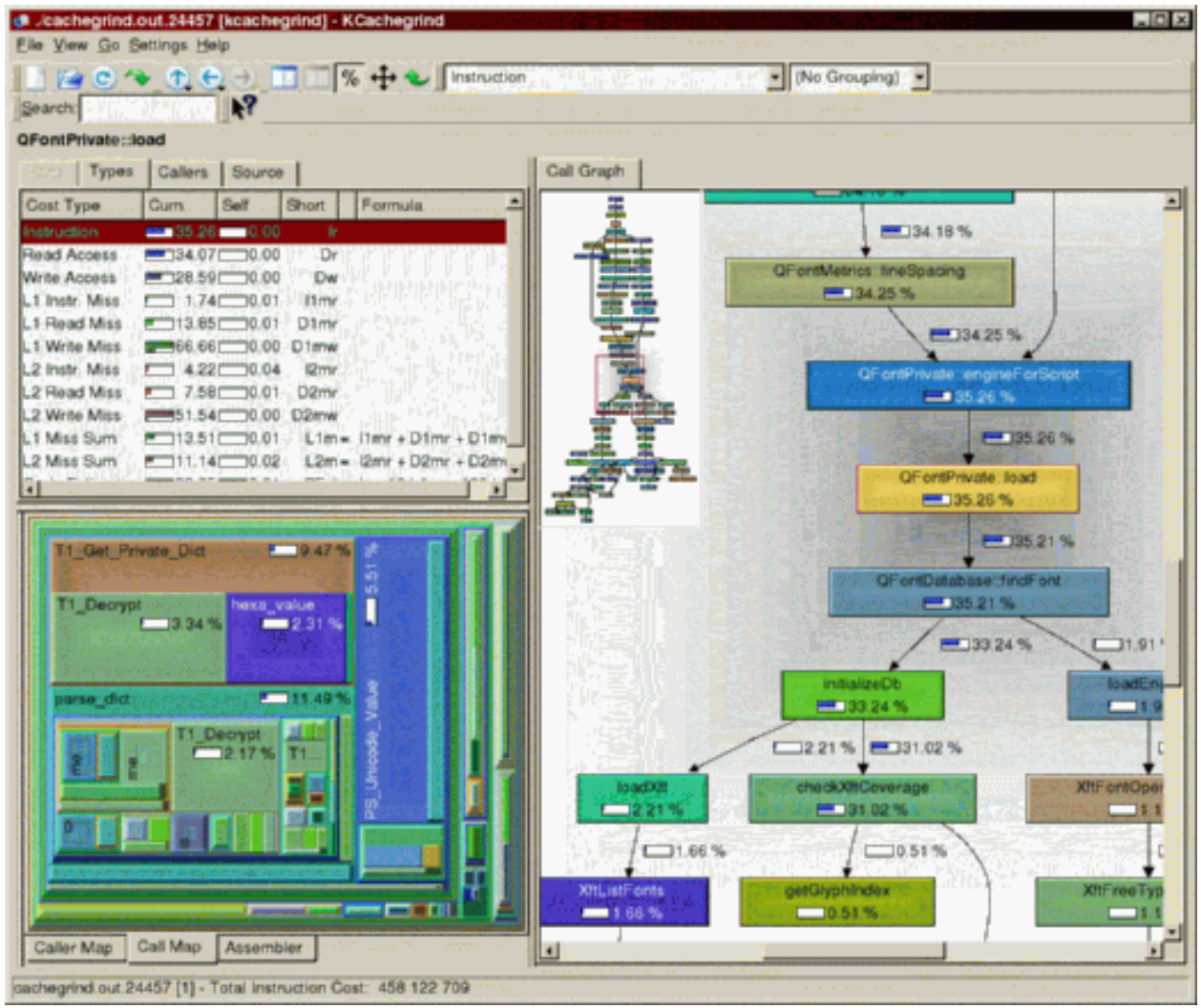
Memcheck

Uses Valgrind to:

- **Use of uninitialised memory**
- **Reading/writing memory after it has been free'd**
- **Reading/writing off the end of malloc'd blocks**
- **Reading/writing inappropriate areas on the stack**
- **Memory leaks -- where pointers to malloc'd blocks are lost forever**
- **Passing of uninitialised and/or unaddressible memory to system calls**
- **Mismatched use of `malloc/new/new[]` vs `free/delete/delete []`**
- **Overlapping `src` and `dst` pointers in `memcpy()` and related functions**
- **Some misuses of the POSIX pthreads API**

KCachegrind

Profiling and cache simulation tool based on Valgrind



Purify

Reed Hastings and Bob Joyce. "*Purify: Fast detection of memory leaks and access errors*" In Proc. 1992 Winter USENIX Conference, pages 125--136, 1992

Commercialized by Rational Software, acquired by IBM

- **Binary translation based verification system with high level program development extension (project management)**
- **Earlier versions used in 15-211 (1997)**
- **Pro: Very mature, powerful tool**
- **Con: Costly, limited range of supported platforms**

- **Commercial competitor: Insure++ from Parasoft**

Profiling

Where is your program spending its CPU time?

Profiling is used to find performance bugs and to fine-tune program performance.

Principle approaches:

- **Compile time instrumentation (gcc –p …)**
- **Statistical sampling (DCPI for Alpha based machines)**
- **Instrumentation via binary translation tools**

gcc –pg …

Add instrumentation (counters) at function granularity (calls to `mcount( )`)

```
[agn@char src]$ gprof driver gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.03     15.71    15.71        51   308.04   308.04  naive_kernel
 14.11     20.14     4.43 88358912     0.00     0.00  is_alive
  7.58     22.52     2.38        20   119.00  1570.00  run_benchmark
  6.02     24.41     1.89 11044258     0.00     0.00  s_buf1_set
  5.76     26.22     1.81 11044258     0.00     0.00  s_buf_set
  5.67     28.00     1.78        51    34.90    34.90  nofunc8_next_generation
  3.18     29.00     1.00 11044258     0.00     0.00  naive_set
  2.29     29.72     0.72        51    14.12    14.12  s_buf_kernel
  2.10     30.38     0.66 11044258     0.00     0.00  nofunc5_turn_on
. . .
```

Another example

```
int f1(int x)
{
  int y,z;
  for (y=1,z=0; y<=x; y++)
    z = z+y;
  return z;
}
int f2(int x)
{
  int y,z;
  for (y=1,z=0; y<=x; y+=2)
    z = z+y;
  return z;
}
int main()
{
  int a,b,c;

  for (a=1000; a<30000; a++)
  {
    b = f1(a);
    c = f2(a);
  }
  return 0;
}
```

```
$ gcc –pg example.c –o example.exe
$ ./example.exe
$ gprof example.exe

Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds  calls  us/call  us/call  name
 68.18    1.95     1.95    29000    67.24    67.24  f1
 31.82    2.86     0.91    29000    31.38    31.38  f2
```

Debugging an Entire System?

Debugging kernel level code is hard: mistakes generally crash the system. Real-time constraints prevent seting breakpoint is places like interrupt handlers or I/O drivers.

Alternatives:

- **SimOS (Stanford, http://simos.stanford.edu/) defunct**
- **Vmware: commercial version of SimOS for virtualizing production server, running Windows under Linux or vice versa**
- **Simics: commercial system level simulation for computer architecture research and system level software development**
- **User Mode Linux: Run Linux under Linux as a user level process http://user-mode-linux.sourceforge.net/**
- **Xen: Hypervisor provides virtualized machine for kernel to run on: http://www.cl.cam.ac.uk/Research/SRG/netos/xen/index.html**
- **Denali Isolation Kernel: virtualized machine and special guest OS: http://denali.cs.washington.edu/pubs/**

User-Level Linux

User-Mode Linux is a safe, secure way of running Linux versions and Linux processes. Run buggy software, experiment with new Linux kernels or distributions, and poke around in the internals of Linux, all without risking your main Linux setup.

Lego Mindstorms Hardware
Robotics Command EXplorer (**RCX 1.0**)

Hitachi H8/300 microcontroller

16 KB ROM, 16 KB RAM

16 MHz system clock

16 bit free-running timer

3 sensor ports (for light, touch, sound, temperature, etc.)

3 motor (actuator) ports

IR transmitter/receiver

LCD screen

RCX Hardware

Hitachi H8/3292 microcontroller

16-bit Timer

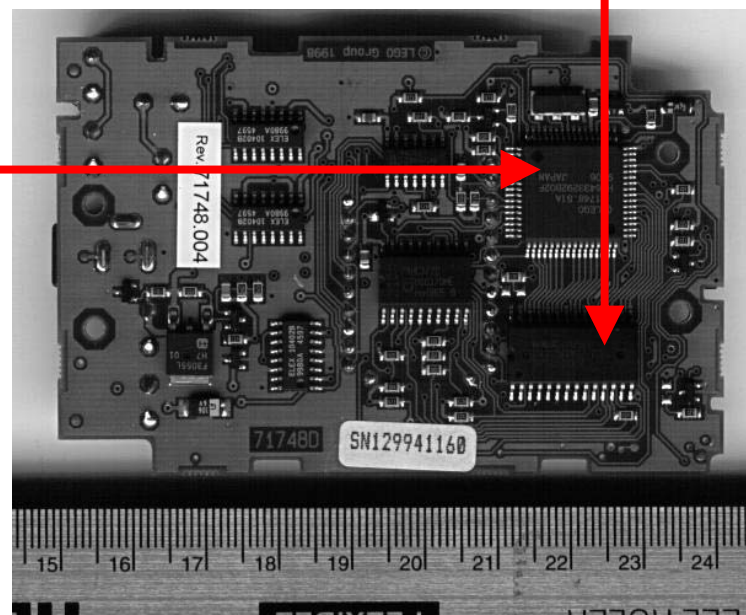
8-bit Timer

I/O Ports

16k ROM

A/D Converter

32k RAM



H8/3292 Microcontroller Details

- **Series H8/3297**
- **Product name H8/3292**
- **Part number HD6433292**
- **ROM size 16K**
- **Internal RAM size 512 bytes**
- **Speed 16MHz @ 5V**
- **8-bit Timers 2**
- **16-bit Timers 1**
- **A/D Conversion 8 8-bit**
- **I/O pins 43**
- **Input only pins 8**
- **Serial port 1**
- **10mA outputs 10**

Hitachi H8/3292 microcontroller

- **16-bit Address Space**
- **8 16-bit General Purpose Registers: R0, R1, R2, … , R7**
  - **Byte-addressable: R0H, R0L, R1H, R1L, … , R7H, R7L**
- **2 Control Registers**
  - **16-bit Program Counter (PC)**
  - **8-bit Condition Code Register (CCR) (status register)**
  - **R7 is used as the Stack Pointer (SP = R7)**
- **16 kb ROM**
  - **contains a driver that is executed when the RCX is powered**
  - **provides low-level routines that call interrupt handlers in RAM so that interrupt handling can be customized in firmware**
- **512 b Static RAM on-chip (mapped to 0xFD80-0xFF7F)**
- **Factory default firmware (standard firmware) contains a bytecode interpreter, which leaves only 6 kb of RAM for user programs ☹. Idea: replace the factory default firmware with our own lean and mean firmware ☺.**

CPU Registers

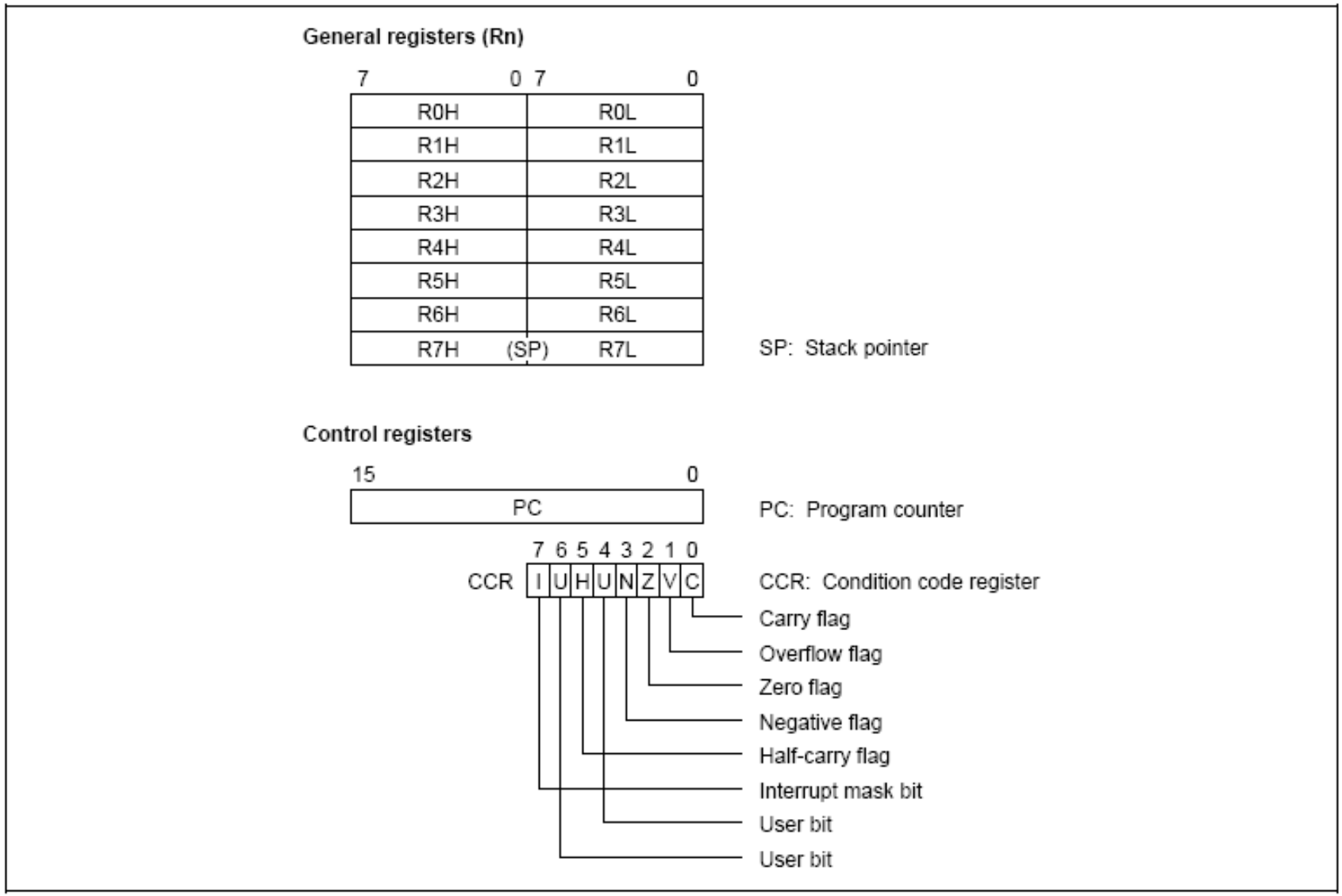


Figure 2-1 CPU Registers

CPU Instructions

The H8/300 CPU has 57 types of instructions, which are classified by function in table 2-3.

Table 2-3 Instruction Classification

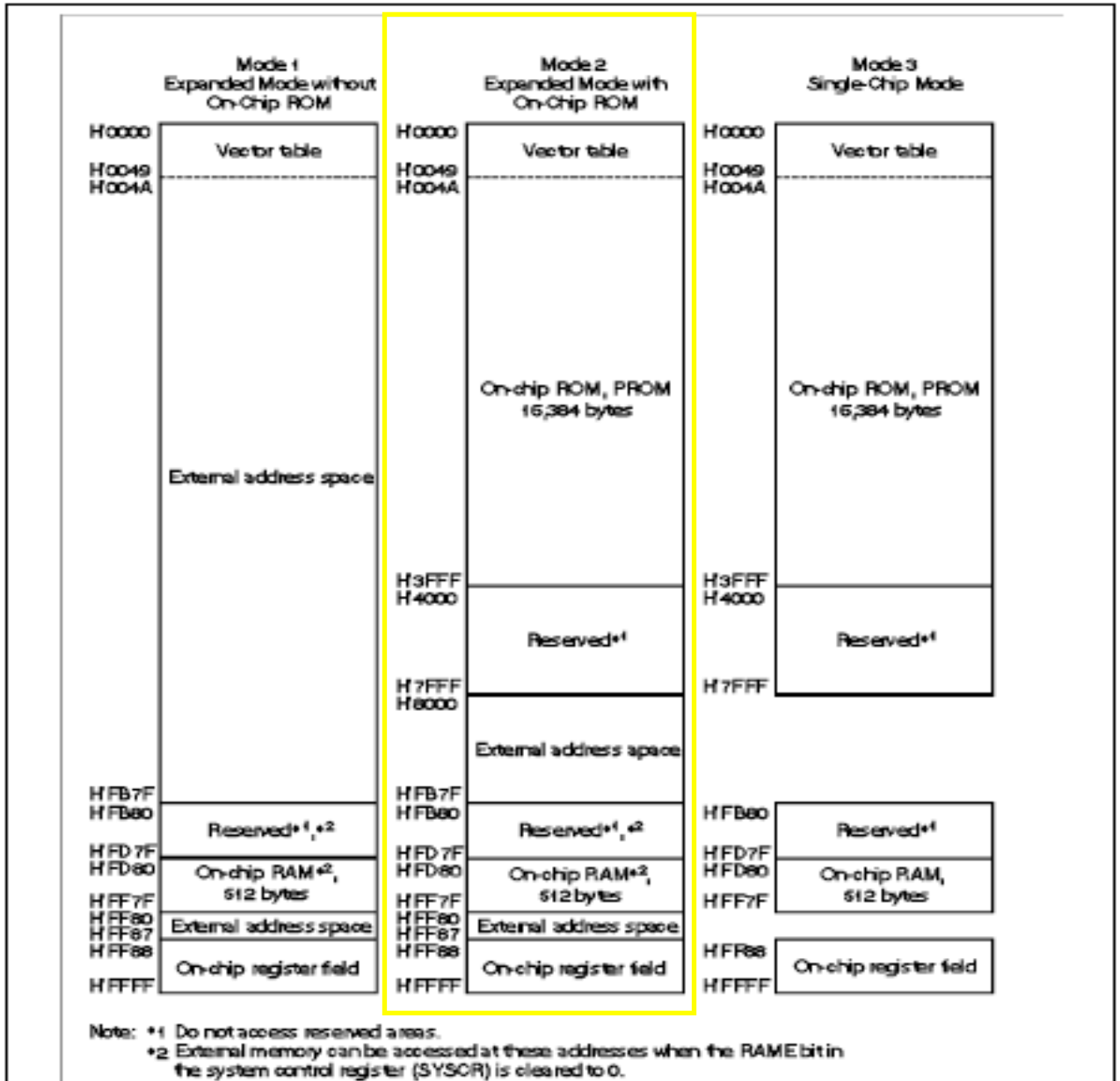
Function	Instructions	Types
Data transfer	MOV, MOVTPE*3, MOVFPE*3, PUSH*1, POP*1	3
Arithmetic operations	ADD, SUB, ADDX, SUBX, INC, DEC, ADDS, SUBS, DAA, DAS, MULXU, DIVXU, CMP, NEG	14
Logic operations	AND, OR, XOR, NOT	4
Shift	SHAL, SHAR, SHLL, SHLR, ROTL, ROTR, ROTXL, ROTXR	8
Bit manipulation	BSET, BCLR, BNOT, BTST, BAND, BIAND, BOR, BIOR, BXOR, BIXOR, BLD, BILD, BST, BIST	14
Branch	Bcc*2, JMP, BSR, JSR, RTS	5
System control	RTE, SLEEP, LDC, STC, ANDC, ORC, XORC, NOP	8
Block data transfer	EEPMOV	1
		Total 57

Notes: 1. PUSH Rn is equivalent to MOV.W Rn, @–SP.
POP Rn is equivalent to MOV.W @SP+, Rn.
2. Bcc is a conditional branch instruction in which cc represents a condition code.
3. Not supported by the H8/3297 Series.

External 32 kb RAM

- **External RAM is referenced in address range: 0x8000-0xFB7F, only address 0x7B7F = 31,615 bytes.**
- **On-chip RAM (512 bytes) is mapped to: 0xFD80-FF7F.**
- **External RAM in: 0xFF80-0xFF87 (hmmm, 8 bytes ☺).**
- **On-chip registers are mapped to memory in: 0xFF88-0xFFFF.**

H8/3292 Address Map – Mode 2



Note: *1 Do not access reserved areas.
*2 External memory can be accessed at these addresses when the RAME bit in the system control register (SYSCR) is cleared to 0.

Stack Frame Layout

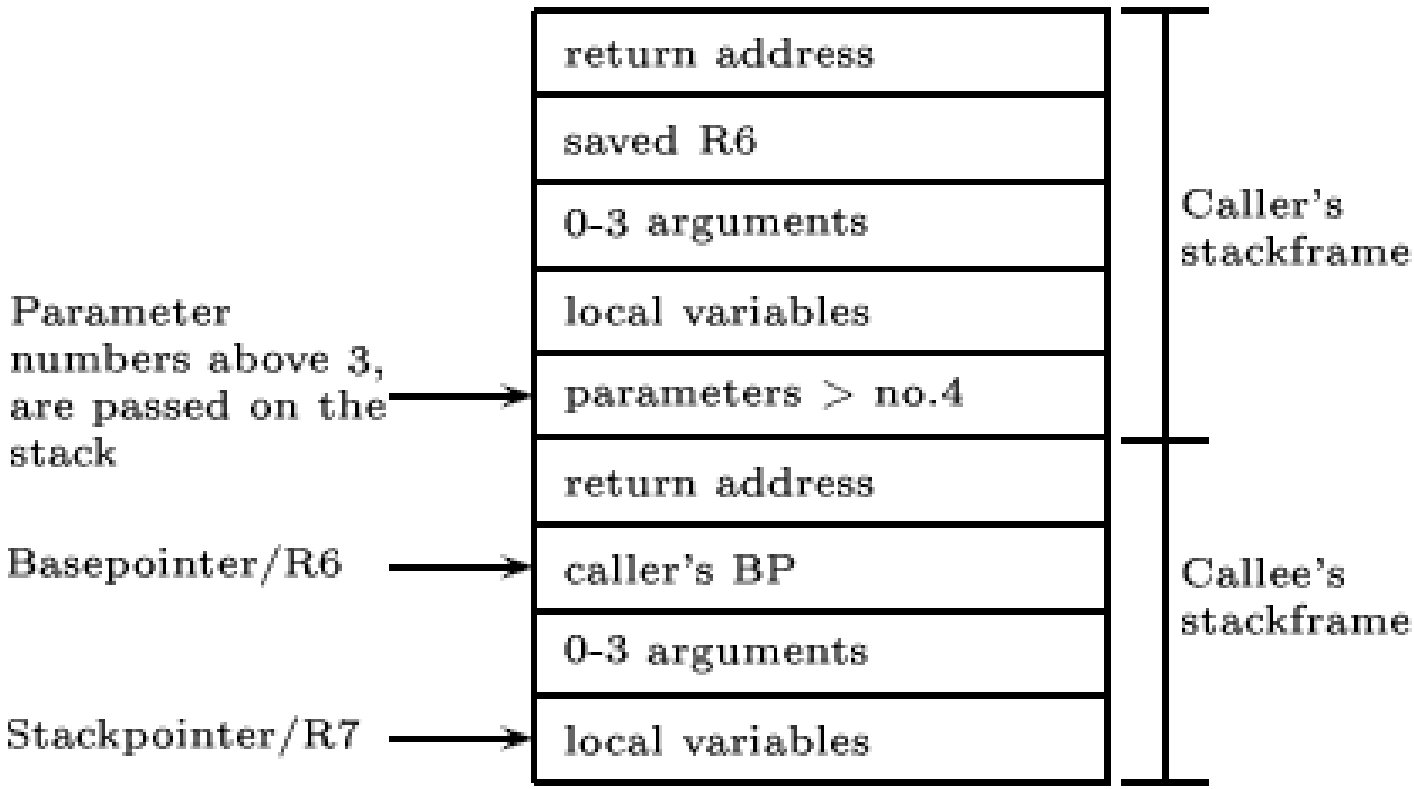


Figure 1: GCC's stack frame layout

BrickOS (LegOS 0.2.4) Kernel

- Kernel Initialization and Timing

  kmain.c and systime.c

- Task Management

  tm.c

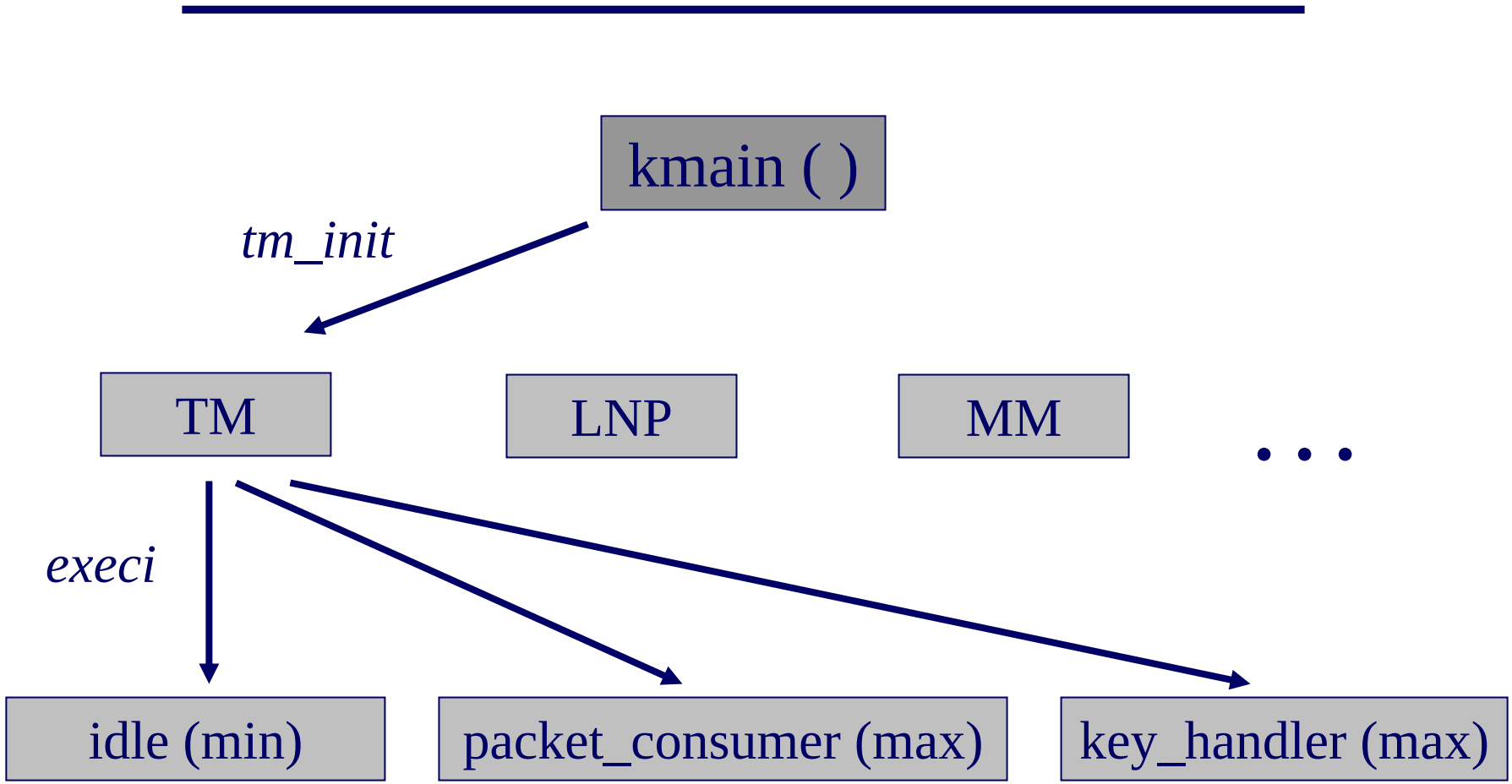
- Interprocess Communication

  lnp.c, lnp-logical.c and semaphore.c

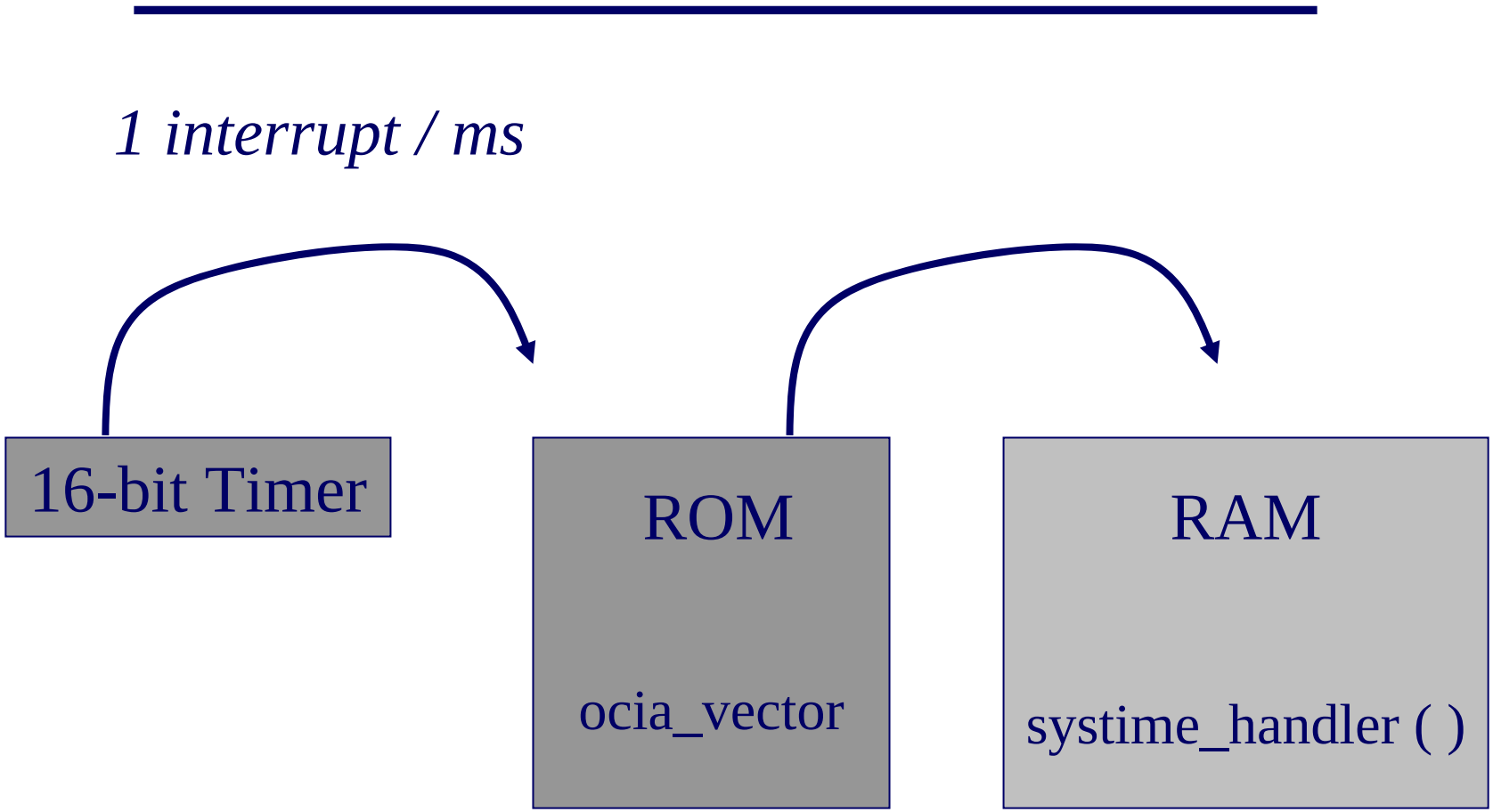
Hitachi H8 ROM

- Start-up driver for Firmware = kmain( )
  - The kernel starts when kmain( ) is called by ROM.
  - This function initializes the kernel before starting in either single tasking or multitasking mode.
  - In multitasking mode, 3 tasks are started:
    - idle task (lowest priority)
    - packet_consumer to handle IR-port data (highest priority)
    - key_handler to handle button activity on RCX brick (highest priority)

BrickOS Startup



Timer Interrupts



systime_handler( )

- Increment 16-bit system timer
- Check to see if any events are pending, and call corresponding handler:
  - Motor handler
  - Sound handler
  - LNP checked for timeout
  - …
- **Check whether we need a task switch**

BrickOS Operating System

Formerly, called LegOS.

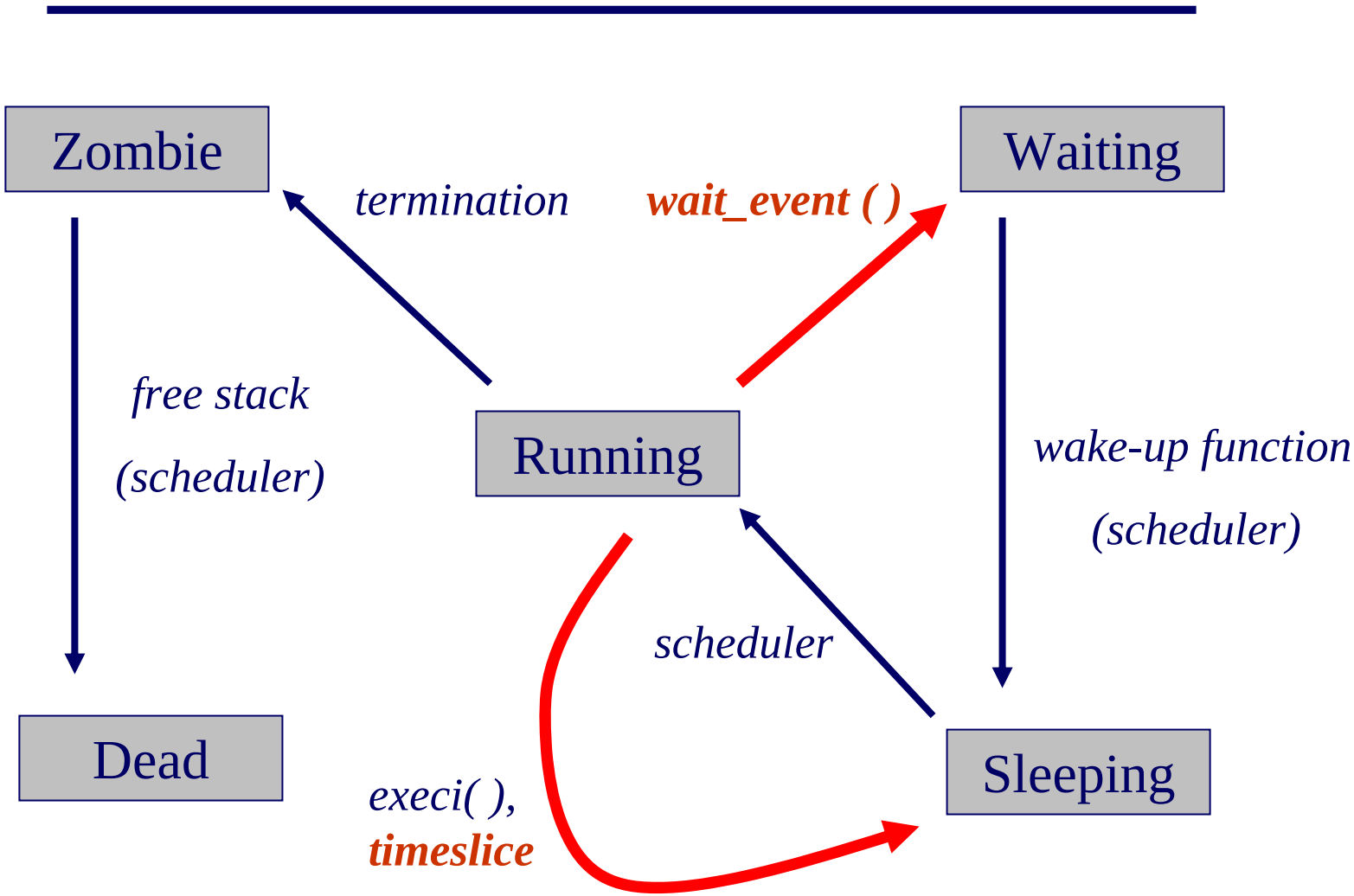
User-level and kernel-level programs are written in C/C++.

Preemptive priority-based round-robin scheduling algorithm with a time quantum of 20 milliseconds.

First fit dynamic memory allocation.

Tasks (threads) in the BrickOS go through states sleeping, running, waiting, zombie, and dead.

The Life of a BrickOS Process



Task Status Constants

T_DEAD = Dead and Gone, Stack Freed

T_RUNNING = Running

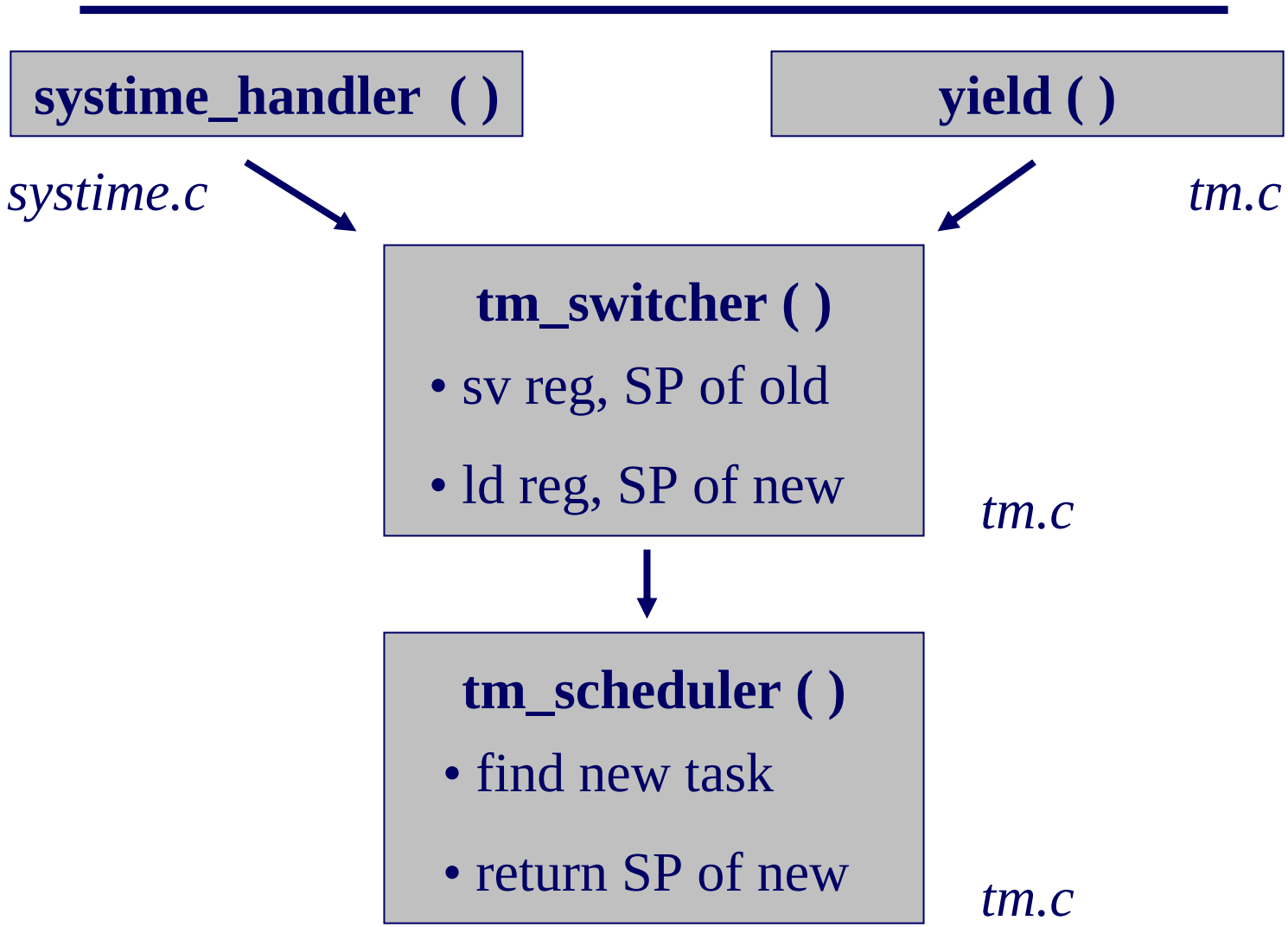
T_SLEEPING = Sleeping, waiting to Run

T_WAITING = Waiting for an Event

T_ZOMBIE = Terminated, Cleanup Pending

T_IDLE = IDLE Task

Scheduler Invocation



BrickOS Tasks

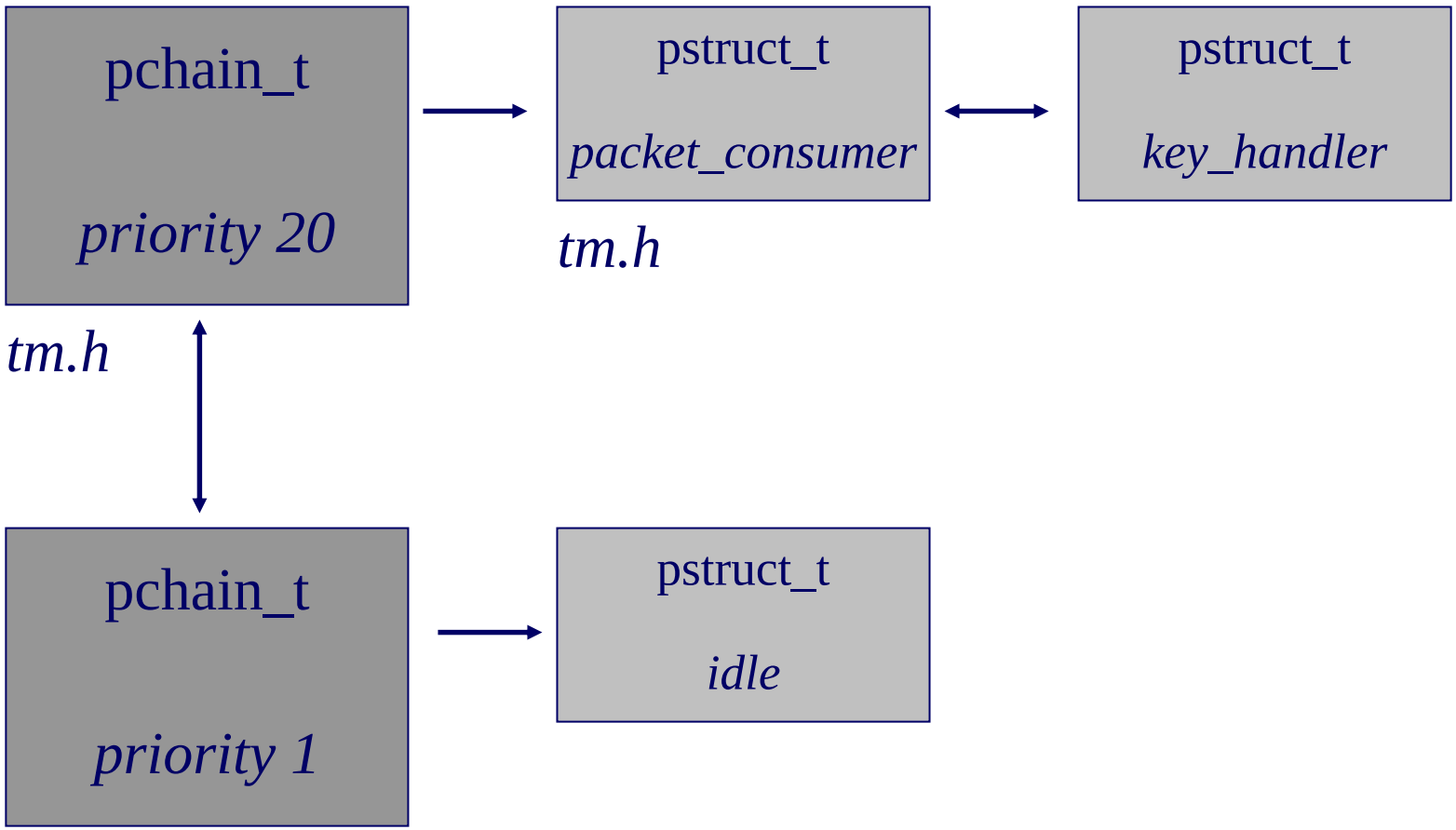
Divided into kernel-level and user-level tasks.

Tasks are sorted into circular queues based on their priorities. Tasks at the same priority level are put into the same circular doubly-linked queue.

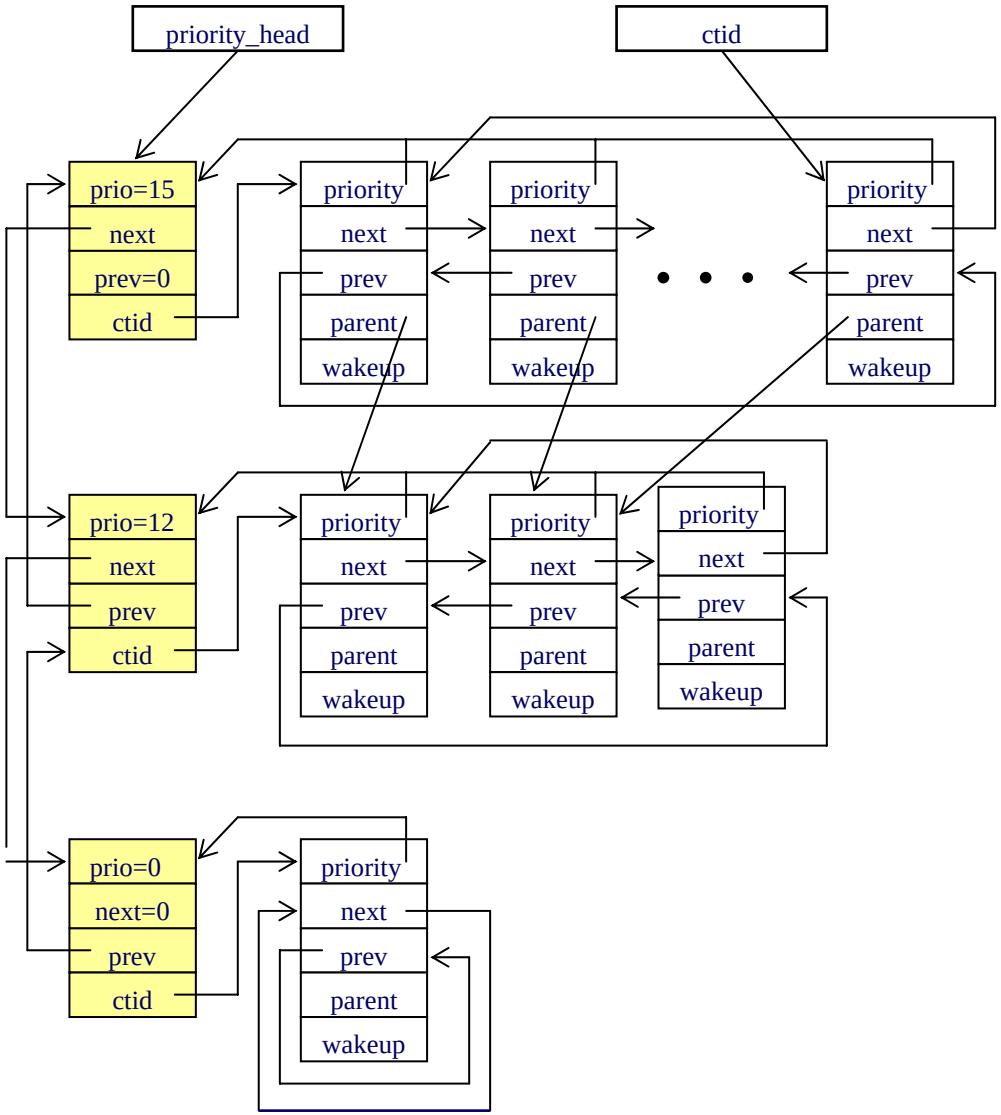
The head of each queue contains a pointer to the previous and next task, the priority level, and the first task in the chain.

A TCB (Task Control Block) **contains the stack pointer, state information, user and kernel flags, and a pointer to the priority chain.**

Prioritized Round Robin w/ Task Queue



Priority Queues



Interprocess Communication

- IR via LegOS Network Protocol (LNP)

  lnp.c and lnp-logical.c

- Semaphores

  semaphore.c

LegOS Network Protocol (LNP)

Two kinds of packets/services:

1) Integrity (“broadcast”)
2) Address (“UDP”)

Three types of communication:

1) PC → RCX
2) RCX → PC
3) RCX → RCX

Integrity Packet

F0: integrity packet id (1 byte)

LEN: length of DATA section (1 byte)

DATA: payload data (0-255 bytes)

CHK: checksum (1 byte)

Addressing Packet

F1: address packet id (1 byte)

LEN: DEST + DATA + SRC (1 byte)

DEST: destination address (1 byte)

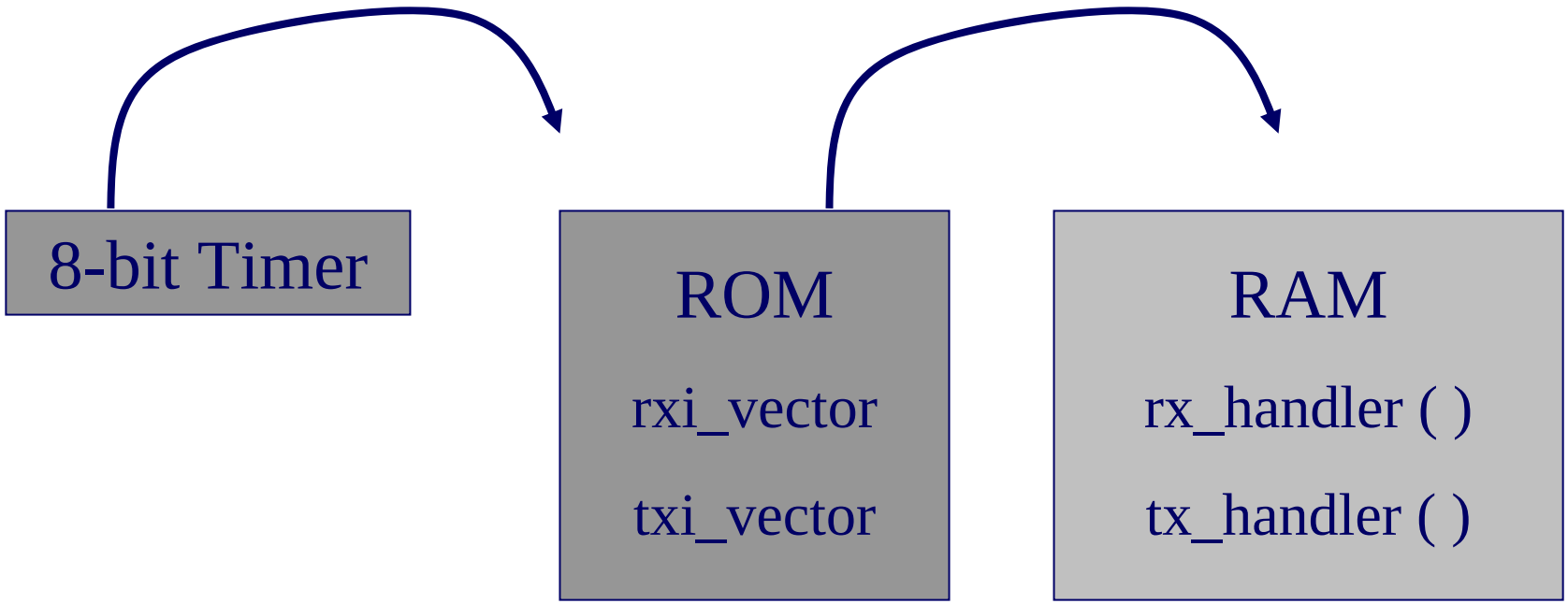
SRC: source address (1 byte)

DATA: payload data (0-253 bytes)

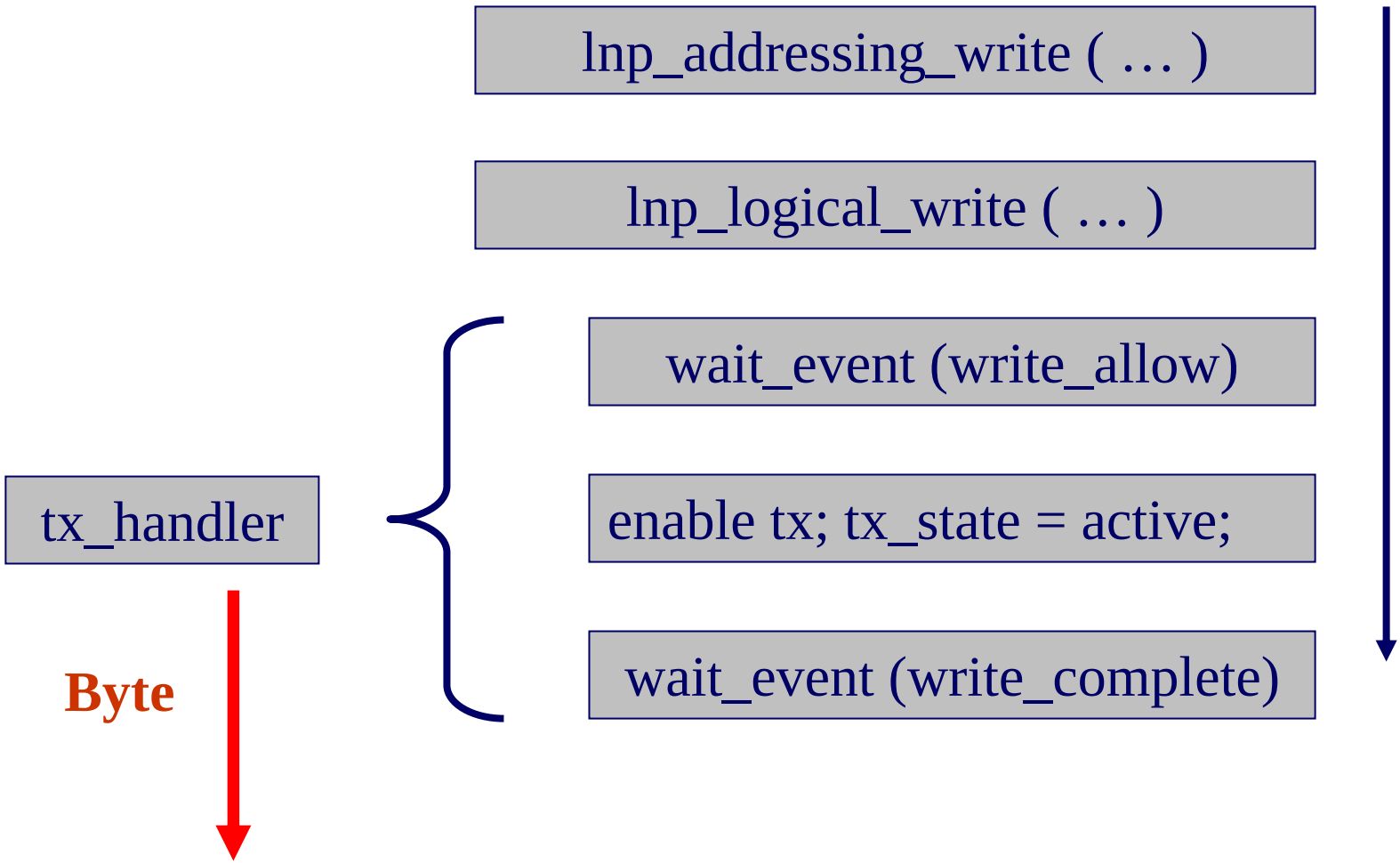
CHK: checksum (1 byte)

LNP Interrupt Driven

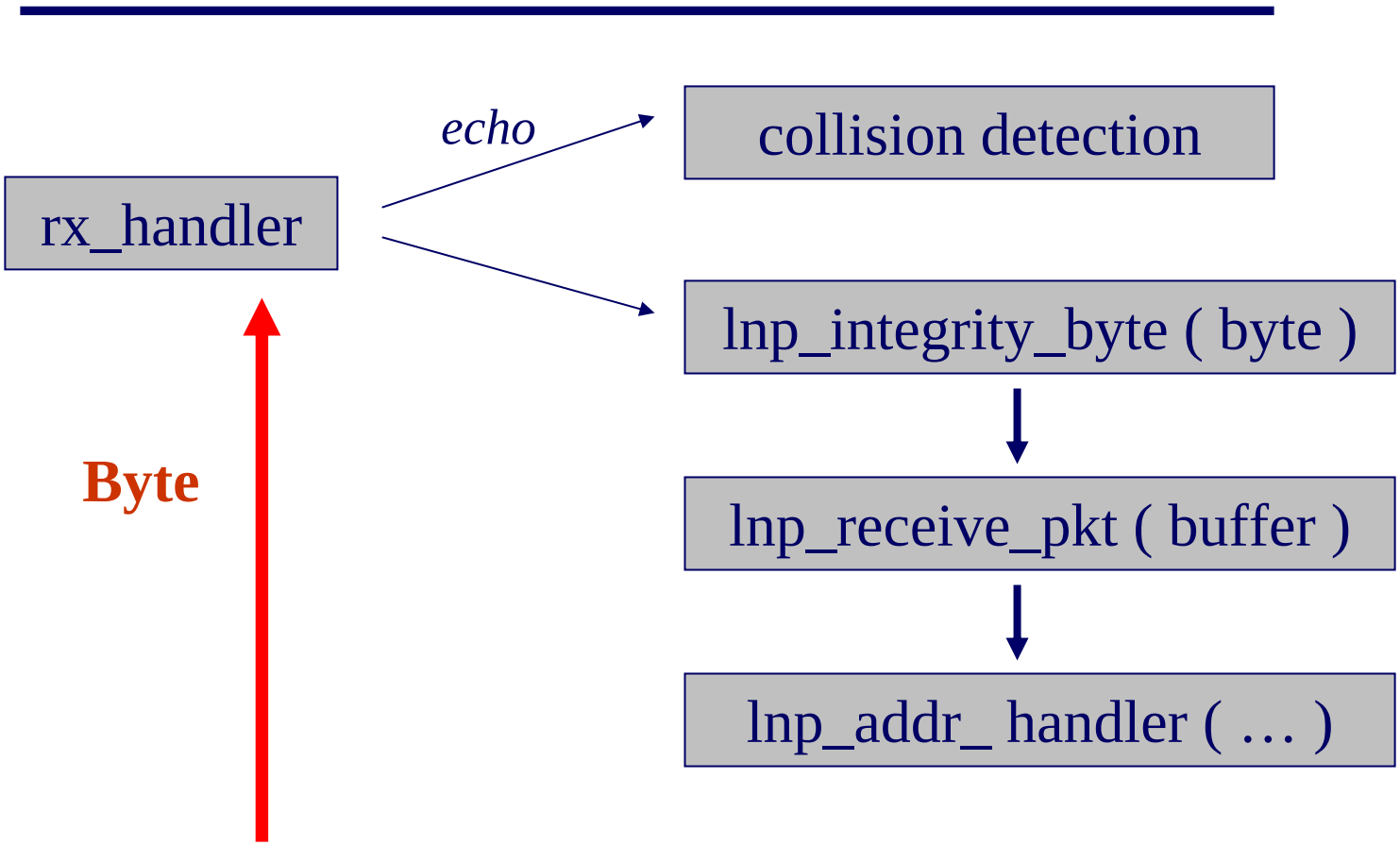
32 interrupts / ms



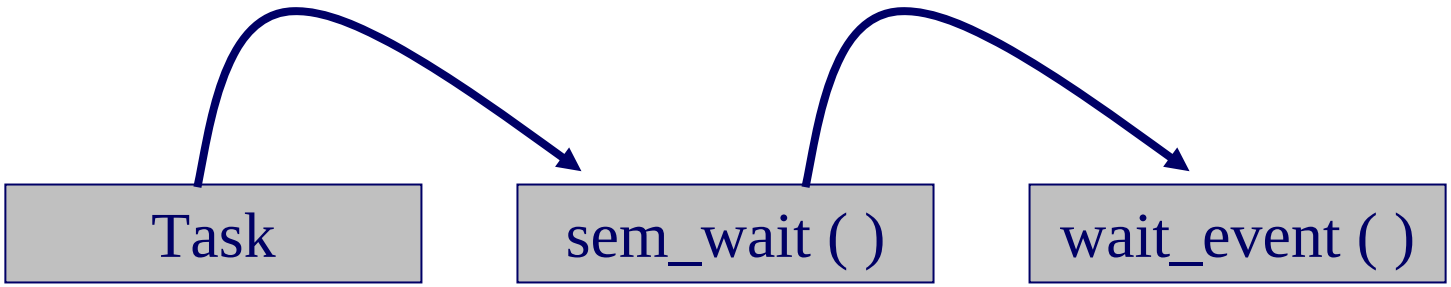
Life of a Packet (Sender)



Life of a Packet (Receiver)



LegOS Semaphores



Kernel Semaphores

tx_sem - only one task can transmit at a time

tm_sem - only one task can touch Task Queue

mm_sem - malloc needs to be memory atomic

BrickOS 0.2.6.10 Installation

http://brickos.sourceforge.net/documents.htm

Windows XP Installation:

- **Install Cygwin**
- **Build the Hitachi-H8 cross compiler (h8300-hitachi-hms-gcc.exe, ..)**
- **Install the brickOS source code and build kernel (brickOS.srec)**
- **Try it:**
  - **Download the kernel: $ ./firmdl3 ../boot/brickOS.srec**
  - **Download a sample application: $ ./dll ../demo/sound.lx**

Linux Installation is similar – brickOS is installed in the Linux Lab.

RCX Internals

http://graphics.stanford.edu/~kekoa/rcx/

Priority-Based Scheduling Algorithm

Each task is given a default time slice (quanta) of 20ms to run before being interrupted by the OS to check if another task is ready to run.

In choosing the next task to run, the OS searches through each task queue, beginning with the highest priority queue, and runs the first READY task found.

Tasks at the same priority level are executed in a fair round-robin fashion.

Task Creation Functions

(int) tid_t execi (&PROCESS_NAME, int argc, char**argv,
priority_t priority, size_t stack_size);

- place function PROCESS_NAME **into the Process queue, returns the Process's assigned Thread ID or -1 if thread failed to start**

Example: execi(&RunMotor, 0, NULL, 1, DEFAULT_STACK_SIZE);
Starts function RunMotor as a thread, with no parameters passed (0, NULL), at the lowest priority (1), with the DEFAULT_STACK_SIZE of 512 being used. The thread ID is passed back, but in this case it is not being stored, but it could have been assigned to a variable type tid_t to keep track of various thread.

void exit (int code);

- exits Process, returning code

Task Termination

void kill (tid_t TID);
Kill Thread associated with (int)tid_t TID **as assigned when it was started by execi ( )**

void killall (priority_t p);
Kill all Processes with a Priority less than p

void shutdown_task (tid_t TID);
Shutdown Thread associated with tid_t TID **wakeup_t wait_event(wakeup_t(*wakeup) (wakeup_t), wakeup_t data); Suspend current Process until Event wakeup function is non-null: unistd.h, tm.c**

Task Wakeup On Event

void yield ( );
-- Yield the rest of the current Task's timeslice

int sleep(int sec);
int msleep(int msec);
-- Pause for an interval of time before executing next commands in current program thread, other program threads will continue to execute commands. Gives up CPU time for other threads.

wakeup_t wait_event (wakeup_t (* wakeup)(wakeup_t), wakeup_t data);
Suspend task until wakeup function returns a non-NULL
Parameters:

- wakeup **the function to be called when woken up**
- data **the wakeup_t structure to be passed to the called function**

Task Priority Levels

Predefined Priority Levels:

PRIO_LOWEST = 1 The Lowest Possible Task Priority

PRIO_NORMAL = 10 The Normal Priority Level

PRIO_HIGHEST = 20 The Highest Task Priority

Example

```
#include <unistd.h>
#include <dbutton.h>
#include <dmotor.h>
int MotorSpeed = 0;

int RunMotor() {
    while (!shutdown_requested())
    {
        motor_a_dir(MotorSpeed);
    }
    return 0;
}

int CheckButton() {
    while (!shutdown_requested())
    {
        if (PRESSED(dbutton(),BUTTON_PROGRAM)) MotorSpeed = 1;
        else MotorSpeed = 0;
    }
    return 0;
}

int main() {
    execi(CheckButton, 0, NULL, 1, DEFAULT_STACK_SIZE);
    execi(RunMotor, 0, NULL, 1, DEFAULT_STACK_SIZE);

    while(!shutdown_requested())
        msleep(1000);

    return 0;
}
```

Setting and Reading Sensors

Must set sensor mode **explicitly before attempting to read sensors.**

void ds_active(volatile unsigned * sensor);
void ds_passive(volatile unsigned * sensor);
-- Set sensor **(possible values: &SENSOR_1, &SENSOR_2, &SENSOR_3) to active or passive type. Light sensor emits light in active mode, rotation modes requires active mode.**

Setting Rotation Sensor

void ds_rotation_on(volatile unsigned * sensor);

void ds_rotation_off(volatile unsigned * sensor);
-- Start/Stop tracking on the Rotation Sensor sensor.

void ds_rotation_set(volatile unsigned * sensor, int pos);
Set Rotation Sensor sensor **to an absolute value** pos**, the rotation sensor should be stationary during the function call Reading Sensor Values (defined Macro) for each sensor pad 1, 2 or 3**

Raw Sensor Input

int SENSOR_1, i**nt** SENSOR_2, **int** SENSOR_3

Light, Touch, and Rotation Sensors

Light Sensor

int LIGHT_1, **int** LIGHT_2, **int** LIGHT_3
Value for light sensor on pads 1, 2, or 3. Scaled to a maximum decoded value of LIGHT_RAW_WHITE using the formula:

(147 - (RAW_LIGHT_READING >> 6) / 7)
Associated Defined Constants:

- **LIGHT_RAW_BLACK = 0xffc0 (active light sensor raw black value)**
- **LIGHT_RAW_WHITE = 0x5080 (active light sensor raw white value)**

Touch Sensor

boolean TOUCH_1, **boolean** TOUCH_2, **boolean** TOUCH_3
Returns value 1=pushed in/on, 0=not pushed/off for a touch sensor on sensor pads 1, 2, or 3

Rotation Sensor

int ROTATION_1, **int** ROTATION_2, **int** ROTATION_3
Rotation Sensor reading

Sensors (cont.)

Activate or Passivate All Sensors

DS_ALL_ACTIVE
  -- Macro to set all Sensors ACTIVE: dsensor.c

DS_ALL_PASSIVE
  -- Macro to set all Sensors PASSIVE: dsensor.c

Battery Readings

int get_battery_mv();
  Get Battery level in XXXX mV: battery.h, battery.c

int BATTERY
  Raw Battery Voltage level: dsensor.h

Controlling Motors

Set Motor Direction

void motor_a_dir(enum MotorDir)

void motor_b_dir(enum MotorDir)

void motor_c_dir(enum MotorDir)
-- The direction MotorDir **is enumerated as: off/freewheeling = 0, fwd = 1, rev = 2, brake = 3 Set the motor Speed**

void motor_a_speed(int speed)

void motor_b_speed(int speed)

void motor_c_speed(int speed)
-- Set Motor to speed **a value between 0-255 Defined Constants:**

- **MAX_SPEED = 255 Constant for upper limit of motor speed**
- **MIN_SPEED = 0 Constant for lower limit of motor speed**

LCD Display

Digit display positions are enumerated from right to left 0 to 5.

void cls ();
-- Clear user portion of screen

void lcd_unsigned(unsigned int u);
-- Display unsigned value u **in decimal, position 0 not used.**

void lcd_digit(int i);
-- Display single digit of integer i **at position 0 (right of the man symbol)**

void lcd_clock(int i);
-- Displays i **with the format XX.XX**

void lcd_number (int i, lcd_number_style n, lcd_comma_style c);
-- Displays integer i **with the following characteristics:**

void delay (unsigned ms);
-- Set Display Delay to approximately ms **mSec**

void cputs(char * s);
-- Write string s **to LCD (only forst 5 characters)**

void cputw(unsigned word);
-- Write a HEX word **to LCD (only forst 5 characters)**

void cputc (char c, int pos);
-- Write ASCII character to specified position of LCD. (this is essentially a dispatcher for cputc_[0-5] functions)

Reading RCX Buttons

Functions to directly read the state (debounced) of the 4 RCX Control buttons

int getchar( )
-- Wait for a Keypress and return the Key Code

wakeup_t dkey_pressed (wakeup_t data)
-- Wakeup if any of the keys is pressed

wakeup_t dkey_released (wakeup_t data)
-- Wakeup if any of the keys is released Current Key activity can also be derived by checking one of the following variables:

volatile unsigned char dkey
-- The current single key pressed

volatile unsigned char dkey_multi
-- The currently active keys, multiple keys readable as a bitmask.

Reading RCX Buttons (cont.)

Key Macros

KEY_ONOFF (0x01) the on/off key is pressed
KEY_RUN (0x02) the run key is pressed
KEY_VIEW (0x04) the view key is pressed
KEY_PRGM (0x08) the program key is pressed
KEY_ANY (0x0f) any of the keys

To Read the Raw status of the RCX Control buttons:

int dbutton(void);
Get Button States, note: this function does not return a de-bouced output, better to use the functions from dkey.h listed above.
Macros for polling the state of these buttons:

RELEASED(state, button) **((state) & (button))**
True if any of the specified buttons is released

PRESSED(state, button) **(!RELEASED(state,button))**
True if all of the specified buttons are pressed BUTTON_ONOFF (0x0002) the on/off button
BUTTON_RUN (0x0004) the run button
BUTTON_VIEW (0x4000) the view button
BUTTON_PROGRAM (0x8000) the program button

Playing Sounds

void dsound_system (SOUND);
-- Play Pre Defined System SOUND:Ê DSOUND_BEEP

int dsound_finished ( );
--Returns a Non-Zero if sound has finished playing.: dsound.h

int dsound_playing ( );
-- Returns nonzero value if a sound is playing: dsound.h

void dsound_stop ( );
-- Stop playing current sound/song

void dsound_play(const note_t *notes);
-- Plays notes **an array of note_t as defined below:**

```
typedef struct {
    unsigned char pitch; //!< note pitch, 0 ^= A_0 (~55 Hz)
    unsigned char length; //!< note length in 1/16ths
} note_t;
```

Pre Defined Note Lengths: WHOLE, HALF, QUARTER, EIGHTH
Pre Defined Pitches (Octave X **= 0-7):**

PITCH_AX**, PITCH_Am**X**, PITCH_C**X**, PITCH_Cm**X**, PITCH_D**X**, PITCH_Dm**X**, PITCH_E**X**, PITCH_F**X**, PITCH_Fm**X**, PITCH_G**X**, PITCH_Gm**X**, PITCH_H**X**, PITCH_END, PITCH_MAX, PITCH_PAUSE**

Example: Brick Sorting Problem

Sort different colored 2x2 Lego bricks into bins containing bricks of the same color.

Fix the maximum number of colors to be sorted.

Limitations:

- **Hardware: Color sensor in RIS 2.0**
- **Software: Limited memory available for program and data**

Example: 6-colored brick sorter (requires 3 engines): http://www.philohome.com/bricksorters/sorter3.htm