Inicialización del kernel del MINIX 2.0: main.c, start.c y table.c

Enrique González Cabrera
David J. Hernández Cerpa

Introducción

En la fase de inicialización del sistema, existen múltiples transferencias de control entre rutinas en lenguaje ensamblador y código en C. Las rutinas en ensamblador están localizadas en el fichero mpx386.s (para máquinas de 32 bits) o el mpx88.s (para las de 16 bits). El código en C se encuentra principalmente en los ficheros start.c y main.c, los cuales hacen uso de rutinas y funciones implementadas en otros archivos. También utiliza estructuras de datos de otros ficheros como el table.c

En esta versión del MINIX (2.0) se ha empleado una considerable cantidad de esfuerzo en reorganizar el código, de modo que pueda ser estudiado más fácilmente y sobre todo para que el código sea independiente de la máquina, de forma que pueda ser más fácilmente transportable a otras plataformas.

Start.c

Este archivo contiene una función principal (**cstart**) y un par de funciones que son llamadas por la misma (**k_atoi** y k_**getenv**). Dentro de estas funciones se encuentra el código de inicialización para Minix en procesadores Intel siguiendo el criterio de separar las funciones dependientes del hardware a favor de una mejor transportabilidad.

El código en ensamblador de la rutina **mpx386.s**, encargada de la inicialización, llama después de una serie de pasos a la función cstart. Esta última se encarga de algunos procesos de inicialización y de llamar a otras funciones que inicializan la tabla de descriptores global, la estructura central de datos usada por los procesadores Intel de 32 bits para controlar la protección de memoria, y la tabla descriptores de interrupción, utilizada para seleccionar el código que se va a ejecutar para cada posible tipo de interrupción.

Lo primero que hace cstart es llamando a la función **prot_init**. Esta función inicializa la tabla de descriptores globales la cuál es utilizada por el procesador para comprobar la seguridad en los accesos a memoria. Además inicializa la tabla de descriptores de interrupciones que se utiliza cuando se producen éstas.

A continuación realiza tareas como copiar los parámetros de arranque a la parte de memoria asignada para el kernel, realizando una conversión a valores numéricos mediante la función **k_atoi**. Además, determina el tipo de tarjeta de vídeo, el tamaño de la memoria, el tipo de procesador, el modo de operación (real o protegido)–estas últimas operaciones se hacen mediante llamadas a función **k_getenv-.**

Por último comprueba si se puede retornar al monitor y actualiza la dirección de retorno en consecuencia (estos valores serán utilizados por el mpx386.s).

Las otras dos funciones del fichero son funciones auxiliares utilizadas en start():

- k_getenv: Función a la que se le pasa una variable de entorno "name" y devuelve un puntero a la cadena con el valor de la variable.

- k_atoi: Función que convierte ristra a entero.

Main.c

Éste fichero contiene la siguiente función que es llamada por mpx386.s después de cstart, y se encarga de completar la inicialización y de comenzar la ejecución normal del sistema.

Las funciones que se implementan en éste fichero son los siguientes :

- main(): Establece el vector de interrupciones, la tabla de procesos para tareas y servidores, interpreta los tamaños de memoria y planifica la ejecución de las tareas para que se ejecuten por sí mismas.
- panic(s,n): Procedimiento invocado por el sistema cuando se produce una situación que le hace imposible continuar.

main()

Comienza con una llamada a **intr_init**, que configura el hardware de control de interrupciones. Este proceso ha sido separado debido a que es dependiente del hardware, y se le pasa un parámetro de configuración para indicar si se han de inicializar las interrupciones para Minix o reinicializarlas a su estado original. Esta función además se asegura de que ninguna interrupción que tenga lugar durante el proceso de inicialización tenga efecto. Esto se consigue en dos pasos: primero, se envía un byte a cada controlador de interrupciones que inhibe la respuesta a entradas del exterior; segundo, todas las entradas de la tabla de interrupciones utilizadas para acceder manejadores específicos de dispositivos se rellenan con la dirección de una rutina que imprimirá un mensaje si se recibe alguna interrupción espúrea. Estos vectores se volverán a rellenar con sus valores originales uno a uno, a medida que las tareas de entrada/salida ejecutan sus propias rutinas de inicialización.

A continuación se llama a **mem_init**, función que inicializa un array que define la localización y tamaño de cada porción de memoria restante en el sistema. Al igual que para la función anterior, esta función es dependiente del hardware, y por ello ha sido aislada en un archivo separado para mantener al main libre de código no transportable.

La siguiente tarea que realiza el main consiste en inicializar la tabla de procesos, marcandolos como libres y asignándoles un número identificativo. Acto seguido pasa a inicializar sus entradas para tareas del sistema y servidores, con la información necesaria para que cuando las primeras tareas y procesos sean tratados, su mapeo de memoria y registros sean asignados correctamente.

La tabla de procesos es un vector en el cual en cada posición se almacena información necesaria para que los procesos puedan ser ejecutados. Sus campos están descritos más adelante.

En esta etapa, se inicializa la tabla de procesos con la información necesaria para ejecutar las tareas, servicios y el init. Se inicializan los punteros de los segmentos de código, datos y pila para cada una de las tareas del sistema, junto con el MM (memory manager), FS (File system) y el INIT. Estas tareas deben de estar presentes en el arranque del sistema y ninguna de ellas terminará durante la correcta operación del mismo.

Todas las tareas están compiladas dentro del mismo fichero que el kernel, y la información acerca de sus requerimientos de pila está en un array llamado *tassktab*, definido en **table.c** . Dado que todas las tareas están compiladas dentro del núcleo y pueden utilizar código y acceder a datos localizados en cualquier lugar del kernel, el tamaño de una tarea individual no es relevante, y el campo de la estructura del proceso designado para indicar su tamaño no es significativo, pues se utilizará toda la porción de código del kernel.

A continuación se le asigna a cada servidor su localización en memoria. Para ello el sistema cuenta con un vector denominado **sizes**. En este se encuentra la información del tamaño del código y datos de los procesos. Este tiene al menos ocho elementos, de manera que los dos primeros dicen el tamaño del código y datos del **KERNEL** (que será utilizado para cada una de las tareas), los dos siguientes los tamaños respectivos al **MM**, y de la misma manera con el **FS** y el **INIT.** Si se tiene activado el servicio de red, existirán dos posiciones más.

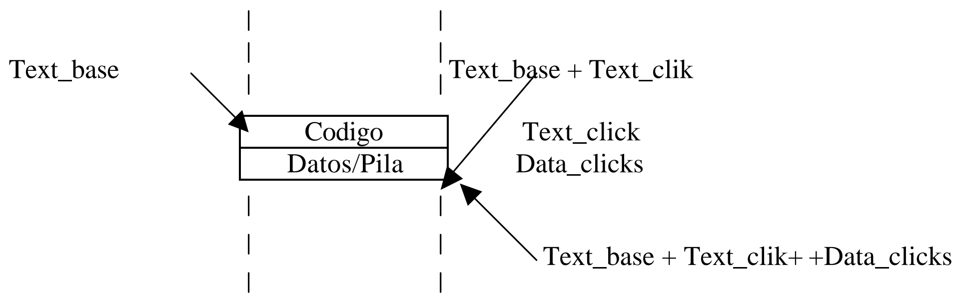
sizes(0) = tamaño del código del Kernel.
sizes(1) = tamaño de los datos del Kernel.
sizes(2) = tamaño del código del manejador de memoria (MM).
sizes(3) = tamaño de los datos del manejador de memoria (MM).
sizes(4) = tamaño del código del File System (FS).
sizes(5) = tamaño de los datos del File System (FS).
sizes(6) = tamaño del código de INIT.
sizes(7) = tamaño de los datos de INIT.

Estructura del vector “sizes” (tamaño en clicks, 1 click=256 bytes).

La utilización de este vector posibilita la modificación del los procesos del sistema (kernel,MM,FS,INIT), sin que esto suponga problema alguno en la recompilación del núcleo, de modo que no es necesaria la modificación del main en tales casos.

Para cada proceso se mapea memoria para código,datos y pila de la siguiente forma:

Como ya hemos comentado las tareas poseen todas la misma posición base y tamaño, y que es la del kernel. El segmento de datos es situado a continuación del segmento de código y la posición de la pila está a continuación de ésta ocupando la misma zona de memoria



Si estamos compilando para la versión de Minix de 32 bit, los servicios y el init del sistema serán situados por encima del rango de 1MB de memoria. Esto permitirá que el sistema de ficheros utilice un amplio bloque de caché de disco dentro de la memoria convencional.

Los procesos IDLE y HARDWARE requieren un tratamiento especial. El proceso IDLE es un bucle que no hace nada y que se ejecuta cuando no hay ningún otro proceso en ejecución, y el proceso HARDWARE existe para propósitos internos (bookkeeping, relacionado con el manejo de interrupciones). Todos los demás procesos se colocan en las colas apropiadas mediante la función **lock_ready(rp)**, que se encarga de colocar el proceso indicado en la cola correspondiente al mismo.

El último paso en la tarea de inicialización de cada elemento de la tabla de procesos es llamar a la función **alloc_segments**, la cual es otra función dependiente de la máquina que asigna a sus campos correspondientes la localización, tamaño y nivel de permiso para los segmentos de memoria usados por cada proceso. Para versiones antiguas del procesador Intel que no soportan el modo protegido (como el i8088 y i8086), sólo determina los segmentos de memoria (obviando la asignación de permisos).

A continuación se indica (en la variable bill_ptr) qué proceso va a ser enviado al procesador. Debido a que tiene que tener un valor inicial, se le asigna la dirección del proceso IDLE. Más adelante será modificado por la siguiente función llamada, **lock_pick_proc**. Se encargará de realizar esta tarea (modificar el proceso que está activo en el procesador) y hará que la variable*proc_ptr* apunte a la entrada en la tabla de procesos del próximo proceso a ejecutar, lo cual realiza examinando las colas de procesos de tareas, servidores y usuarios, en ese orden. En esta parte de la inicialización el puntero *proc_ptr* contendrá la entrada para la tarea de consola, la cual es la primera en ser ejecutada.

Por último, pasamos el control a la rutina **restart**, encargada del manejo de las colas de procesos, momento a partir del cual se puede decir que Minix se está realmente ejecutando.

panic()

Este procedimiento es invocado por el sistema cuando se produce una situación que le hace imposible continuar.

Este tipo de situaciones podrían ser, por ejemplo:

→ Imposibilidad de leer un bloque crítico del disco.
→ Detección de un estado interno incoherente.
→ Llamada de una parte del sistema a otra con parámetros incorrectos.

Table.c

Este fichero es simplemente para definir el vector *tasktab*, que contiene los requerimientos de pila de cada tarea del sistema. Según las tareas que se deseen incluir en el sistema, se compilarán algunas entradas de la tabla de tareas. Esta selección se realiza a través de las macros de compilación codicional.

Definiciones Cabecera Y Estructuras De Datos

/usr/src/kernel/proc.h definiciones referentes a la tabla de procesos. Esta es compartida por el kernel, el mm y el fs.

Cada entrada en la tabla contiene una serie de campos de los cuales main.c hace uso de los siguientes:

p_reg: es una estructura con todos los registros de la máquina. Main inicializa los siguientes:

p_reg.sp = puntero de la pila.
p_reg.pc = contador de programa.
p_reg.psw = palabra de estado (flags).

p_nr: número de proceso.

p_splimit: menor valor permitido para el tamaño la pila.

p_flags: indica el estado de la posición correspondiente en la tabla de procesos.

- Si algún bit = 1 ⇒ el proceso no puede ejecutarse.
- Si P_SLOT_FREE = 1 ⇒ posición de la tabla no está en uso.
- Si NO_MAP = 1 ⇒ se impide que un proceso hijo comience a ejecutarse antes de que se haya creado su mapa de memoria
- El resto de los bits indican bloqueos en espera de recibir o mandar mensajes.

p_map(NR_SEGS): mapa de memoria.

/usr/src/kernel/kernel.h

- principales definiciones para el kernel
- incluye otros ficheros cabecera

/usr/include/minix/config.h definiciones de configuración del MINIX.
/usr/include/minix/const.h constantes globales del MINIX.

/usr/include/minix/type.h tipos de variables a usar por MINIX.
/usr/include/sys/types.h tipos de datos utilizados por el sistema.
/usr/include/limits.h tamaños utilizados en el MINIX.

/usr/include/errno.h errores que se pueden producir.

/usr/src/kernel/const.h

- constantes usadas por el kernel
- algunos vectores de interrupción importantes
- algunos valores necesarios para reprogramar el controlador de interrupciones (8259) después de cada una de éstas.

/usr/src/kernel/type.h tipos en función del chip que tenga la máquina.
/usr/src/kernel/proto.h funciones prototipo.

usr/src/kernel/glo.h variables globales usadas por el kernel.

/usr/include/signal.h señales.

/usr/include/minix/callnr.h se asocian los números de las llamadas al sistema con su nombre.

/usr/include/minix/com.h

- llamadas al sistema
- definiciones relativas a las diferentes tareas (para distinguirlos de los de los procesos, los números de las tareas son negativos).

Macros Y Procedimientos Usados Por Main()

alloc_segments(rp) coloca los segmentos en los procesadores INTEL.

intr_init(n) inicializa el controlador hardware de interrupciones.

isidlehardware(t): pregunta si la tarea t es la tarea ociosa (IDLE) o la HARDWARE.

istaskp(rp) devuelve TRUE si es cualquier tarea menos la ociosa.

lock_pick_proc(rp) Llama a pick_pro(). Decide qué proceso se va a ejecutar a continuación.

lock_ready(rp) Llama a la rutina ready(rp), el cual coloca 'rp' al final de una de las colas de procesos ejecutables. Estas colas son:

TASK_Q	(Mayor prioridad)	Tareas ejecutables
SERVER_Q	(Prioridad intermedia)	MM y FS sólamennte
USER_Q	(Menor prioridad)	Procesos de usuario.

mem_init() Inicializa un array que define la localización y el tamaño de cada porción de memoria disponible en el sistema.

panic(s,n) llamada cuando se ha producido un error irrecuperable.

phys_copy(src,dest,n_bytes) copia un bloque de memoria física. Copiará n_bytes desde src a dest.

restart() habilita las interrupciones e inicia la ejecución de una tarea o proceso.

wreboot() espera a que se pulse una tecla y luego resetea, cargando de nuevo el sistema operativo.

proc_addr macro necesaria porque en C no es posible usar subíndices negativos. El vector proc debería ir desde -NR_TASKS hasta +NR_PROCS. Debe comenzar en 0, de forma que proc[0] se corresponde con la tarea más negativa, y así sucesivamente. Para deducir la posición de la tabla correspondiente a cada proceso, escribimos rp=proc_addr(n), que asigna a rp la dirección de la posición de la tabla para el proceso n, sea éste positivo o negativo.

Algoritmo del main()

[33-42] Declaración de variables.
[45] Inicialización del control de interrupciones.
[48] Mapeo de memoria.
[53-56] Vaciado de la tabla de procesos.
Para cada proceso *hacer*
Liberar su entrada en la tabla.
Fin para
[70-115]Inicializar pila de tareas.
Para cada tarea *hacer*
[73-75] Copiar en la entrada de la tabla de proceso el identificador de la tarea.
Si se trata de una tarea del sistema *entonces*
[77-80] Asignar el puntero de guardia para la pila.
Asignar el puntero de pila.
[83-87] Asignar el mismo puntero base que el del kernel.
Fin si
[90-91] Inicializa el contador de programa y la palabra de estado del programa.
[93-100] Asigna las posiciones de memoria de los punteros de código, datos y pila.
[101] Dejar los índices listos para la siguiente iteración.
[103-104] Marcar la memoria ocupada por el proceso.
[106-115] *Si* no es una tarea del sistema *entonces*
Inicializar el puntero de pila del servidor
Finsi
[117-125] *Si* estamos en modo 386 *entonces*
Cargamos el servidor en memoria extendida
Finsi
Fin para
[133] Introducir el proceso IDLE en la cola de procesos.
[135] Colocar el primer proceso a ejecutar en la cola de procesos.
[136] Pasar el control a la rutina que inicia el funcionamiento normal de Minix.

Nota:los números entre corchetes hacen referencia al número de línea del fuente adjunto.

Cuestiones:

1. ¿Cuáles son las principales funciones que realiza el **main**()? ¿Y el **cstart**()?

2. ¿Qué es la tabla de procesos y para qué se emplea?

3. ¿Qué utilidad tiene el vector *sizes* y qué ventajas proporciona?

Respuestas a las cuestiones

1. Las funciones más importantes del **main** son inicializar el controlador de interrupciones, interpretar los tamaños de memoria, inicializar las entradas en la tabla de procesos para tareas y servidores y pasar el control a las funciones que controlan el funcionamiento normal de Minix.

   El **cstart**, por su parte, inicializa la tabla de descriptores globales (la estructura de datos central usada por los procesadores de 32 bits de Intel para supervisar la protección de memoria) y la tabla de descriptores de interrupciones (usada para referenciar cada porción de código a ejecutar para cada posible tipo de interrupción) y una serie de variables de entorno, tales como el tipo de display, procesador y bus.

2. La tabla de procesos es un vector en el cual en cada posición se almacena información necesaria para que los procesos puedan ser ejecutados, por ejemplo, su pid, el valor de sus registros al hacer un cambio de contexto, etc.

3. El vector size se utiliza para crear el mapa de memoria (segmentos de código, datos y pila) correspondiente a cada proceso o tarea para luego ser incluido en la tabla de procesos. Con este vector obtenemos los tamaños del código y de los datos de los diferentes módulos que componen el sistema operativo (Kernel, mm, fs e init).

   La ventaja principal de este vector es que a la hora de hacer cualquier reestructuración en el tamaño de los archivos que producen los diferentes módulos no tenemos que modificar el núcleo del sistema, ya que este vector se rellena durante el proceso de arranque (no tiene unos valores predeterminados).

START.C

```
#include "kernel.h"
#include <stdlib.h>
#include <minix/boot.h>
#include "protect.h"

PRIVATE char k_environ[256];                          /* cadenas de entorno pasadas por el
cargador */
FORWARD _PROTOTYPE( int k_atoi, (char *s) );

PUBLIC void cstart(cs, ds, mcs, mds, parmoff, parmsize)
U16_t cs, ds;
U16_t mcs, mds;
U16_t parmoff, parmsize;
{

 register char *envp;
 phys_bytes mcode_base, mdata_base;
 unsigned mon_start;

 code_base = seg2phys(cs);                            /* localización del kernel y el monitor */
 data_base = seg2phys(ds);
 mcode_base = seg2phys(mcs);
 mdata_base = seg2phys(mds);

 prot_init();                                         /* inicializa descriptores */

 if (parmsize > sizeof k_environ - 2) parmsize = sizeof k_environ - 2;
 phys_copy(mdata_base + parmoff, vir2phys(k_environ), (phys_bytes) parmsize);   /* copia las variables  de entorno a memoria
de kernel */

 boot_parameters.bp_rootdev = k_atoi(k_getenv("rootdev"));      /* convierte variables de entorno a
parámetros de arranque */
 boot_parameters.bp_ramimagedev = k_atoi(k_getenv("ramimagedev"));
 boot_parameters.bp_ramsize = k_atoi(k_getenv("ramsize"));
 boot_parameters.bp_processor = k_atoi(k_getenv("processor"));

 envp = k_getenv("video");                            /* obtiene información de entorno que le pasa
el boot monitor*/
 if (strcmp(envp, "ega") == 0) ega = TRUE;
 if (strcmp(envp, "vga") == 0) vga = ega = TRUE;

 low_memsize = k_atoi(k_getenv("memsize"));
 ext_memsize = k_atoi(k_getenv("emssize"));

 processor = boot_parameters.bp_processor;            /* 86, 186, 286, 386, ... */

 envp = k_getenv("bus");
 if (envp == NIL_PTR || strcmp(envp, "at") == 0) {
          pc_at = TRUE;
 } else
          if (strcmp(envp, "mca") == 0) {
                    pc_at = ps_mca = TRUE;
          }

#if _WORD_SIZE == 2
 protected_mode = processor >= 286;
#endif

 if (!protected_mode) mon_return = 0;                 /* actualiza la dirección de retorno*/
 mon_start = mcode_base / 1024;
 if (mon_return && low_memsize > mon_start) low_memsize = mon_start;

}

PRIVATE int k_atoi(s)
register char *s;{
 return strtol(s, (char **) NULL, 10);
}

PUBLIC char *k_getenv(name)
char *name;{
 register char *namep;
 register char *envp;

 for (envp = k_environ; *envp != 0;) {
          for (namep = name; *namep != 0 && *namep == *envp; namep++, envp++);
          if (*namep == '\0' && *envp == '=') return(envp + 1);
          while (*envp++ != 0);
 }
 return(NIL_PTR);
}
```

MAIN.C

```
#include "kernel.h"
#include <signal.h>
#include <unistd.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"


PUBLIC void main(){
 register struct proc *rp;          /* puntero a la tabla de procesos*/
 register int t;
 int sizeindex;
 phys_clicks text_base;
 vir_clicks text_clicks;
 vir_clicks data_clicks;
 phys_bytes phys_b;
 reg_t ktsb;                        /* base de la pila del kernel */
 struct memory *memp;
 struct tasktab *ttp;


 intr_init(1);
 mem_init();

 for (rp = BEG_PROC_ADDR, t = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++t) {
		rp->p_flags = P_SLOT_FREE;
		rp->p_nr = t;                    /* número de proceso desde puntero */
		(pproc_addr + NR_TASKS)[t] = rp;  /* puntero a proceso desde número */
 }

 ktsb = (reg_t) t_stack;
 for (t = -NR_TASKS; t <= LOW_USER; ++t) {
		rp = proc_addr(t);               /* t-ésimo slot de proceso */
		ttp = &tasktab[t + NR_TASKS];    /* t-ésimos atributos de tareas*/
		strcpy(rp->p_name, ttp->name);
		if (t < 0) {
			if (ttp->stksize > 0) {
				rp->p_stguard = (reg_t *) ktsb;
				*rp->p_stguard = STACK_GUARD;
			}
			ktsb += ttp->stksize;
			rp->p_reg.sp = ktsb;
			text_base = code_base >> CLICK_SHIFT;
			sizeindex = 0;
			memp = &mem[0];
		} else {
			sizeindex = 2 * t + 2;
		}
		rp->p_reg.pc = (reg_t) ttp->initial_pc;
		rp->p_reg.psw = istaskp(rp) ? INIT_TASK_PSW : INIT_PSW;

		text_clicks = sizes[sizeindex];
		data_clicks = sizes[sizeindex + 1];
		rp->p_map[T].mem_phys = text_base;
		rp->p_map[T].mem_len  = text_clicks;
		rp->p_map[D].mem_phys = text_base + text_clicks;
		rp->p_map[D].mem_len  = data_clicks;
		rp->p_map[S].mem_phys = text_base + text_clicks + data_clicks;
		rp->p_map[S].mem_vir  = data_clicks;
		text_base += text_clicks + data_clicks;
		memp->size -= (text_base - memp->base);
		memp->base = text_base;
		if (t >= 0) {
			rp->p_reg.sp = (rp->p_map[S].mem_vir +
				rp->p_map[S].mem_len) << CLICK_SHIFT;
			rp->p_reg.sp -= sizeof(reg_t);
		}

#if _WORD_SIZE == 4
		if (t < 0) {
			memp = &mem[1];
			text_base = 0x100000 >> CLICK_SHIFT;
		}
#endif
		if (!isidlehardware(t)) lock_ready(rp);
		rp->p_flags = 0;
		alloc_segments(rp);
 }

 proc[NR_TASKS+INIT_PROC_NR].p_pid = 1;
 bill_ptr = proc_addr(IDLE);
 lock_pick_proc();

 restart();
}
```

```
PUBLIC void panic(s,n)
_CONST char *s;
int n;{
 if (*s != 0) {
		printf("\nKernel panic: %s",s);
		if (n != NO_NUM) printf(" %d", n);
		printf("\n");
  }
  wreboot(RBT_PANIC);
}
```

TABLE.C

```
#define _TABLE

#include "kernel.h"
#include <termios.h>
#include <minix/com.h>
#include "proc.h"
#include "tty.h"

/* The startup routine of each task is given below, from -NR_TASKS upwards.
 * The order of the names here MUST agree with the numerical values assigned to
 * the tasks in <minix/com.h>.
 */
#define SMALL_STACK          (128 * sizeof(char *))

#define    TTY_STACK         (3 * SMALL_STACK)
#define    SYN_ALRM_STACK    SMALL_STACK

#define    DP8390_STACK      (SMALL_STACK * ENABLE_NETWORKING)

#if (CHIP == INTEL)
#define    IDLE_STACK        ((3+3+4) * sizeof(char *))  /* 3 intr, 3 temps, 4 db */
#else
#define    IDLE_STACK        SMALL_STACK
#endif

#define    PRINTER_STACK     SMALL_STACK

#if (CHIP == INTEL)
#define    WINCH_STACK       (2 * SMALL_STACK * ENABLE_WINI)
#else
#define    WINCH_STACK       (3 * SMALL_STACK * ENABLE_WINI)
#endif

#if (MACHINE == ATARI)
#define    SCSI_STACK        (3 * SMALL_STACK)
#endif

#if (MACHINE == IBM_PC)
#define    SCSI_STACK        (2 * SMALL_STACK * ENABLE_SCSI)
#endif

#define CDROM_STACK          (4 * SMALL_STACK * ENABLE_CDROM)
#define AUDIO_STACK          (4 * SMALL_STACK * ENABLE_AUDIO)
#define MIXER_STACK          (4 * SMALL_STACK * ENABLE_AUDIO)

#define    FLOP_STACK        (3 * SMALL_STACK)
#define    MEM_STACK         SMALL_STACK
#define    CLOCK_STACK       SMALL_STACK
#define    SYS_STACK         SMALL_STACK
#define    HARDWARE_STACK    0                /* dummy task, uses kernel stack */


#define    TOT_STACK_SPACE          (TTY_STACK + DP8390_STACK + SCSI_STACK + \
           SYN_ALRM_STACK + IDLE_STACK + HARDWARE_STACK + PRINTER_STACK + \
           WINCH_STACK + FLOP_STACK + MEM_STACK + CLOCK_STACK + SYS_STACK + \
           CDROM_STACK + AUDIO_STACK + MIXER_STACK)


/* SCSI, CDROM and AUDIO may in the future have different choices like
 * WINCHESTER, but for now the choice is fixed.
 */
#define scsi_task      aha_scsi_task
#define cdrom_task     mcd_task
#define audio_task     dsp_task
```

```
/*
 * Some notes about the following table:
 *  1) The tty_task should always be first so that other tasks can use printf
 *     if their initialisation has problems.
 *  2) If you add a new kernel task, add it before the printer task.
 *  3) The task name is used for the process name (p_name).
 */

PUBLIC struct tasktab tasktab[] = {
	{ tty_task,		TTY_STACK,	"TTY"		},
#if ENABLE_NETWORKING
	{ dp8390_task,		DP8390_STACK,	"DP8390"	},
#endif
#if ENABLE_CDROM
	{ cdrom_task,		CDROM_STACK,	"CDROM"		},
#endif
#if ENABLE_AUDIO
	{ audio_task,		AUDIO_STACK,	"AUDIO"		},
	{ mixer_task,		MIXER_STACK,	"MIXER"		},
#endif
#if ENABLE_SCSI
	{ scsi_task,		SCSI_STACK,	"SCSI"		},
#endif
#if ENABLE_WINI
	{ winchester_task,	WINCH_STACK,	"WINCH"		},
#endif
	{ syn_alrm_task,	SYN_ALRM_STACK, "SYN_AL"	},
	{ idle_task,		IDLE_STACK,	"IDLE"		},
	{ printer_task,		PRINTER_STACK,	"PRINTER"	},
	{ floppy_task,		FLOP_STACK,	"FLOPPY"	},
	{ mem_task,		MEM_STACK,	"MEMORY"	},
	{ clock_task,		CLOCK_STACK,	"CLOCK"		},
	{ sys_task,		SYS_STACK,	"SYS"		},
	{ 0,			HARDWARE_STACK, "HARDWAR"	},
	{ 0,			0,		"MM"		},
	{ 0,			0,		"FS"		},
#if ENABLE_NETWORKING
	{ 0,			0,		"INET"		},
#endif
	{ 0,			0,		"INIT"		},
};

/* Stack space for all the task stacks.  (Declared as (char *) to align it.) */
PUBLIC char *t_stack[TOT_STACK_SPACE / sizeof(char *)];

/*
 * The number of kernel tasks must be the same as NR_TASKS.
 * If NR_TASKS is not correct then you will get the compile error:
 *   "array size is negative"
 */

#define NKT (sizeof tasktab / sizeof (struct tasktab) - (INIT_PROC_NR + 1))

extern int dummy_tasktab_check[NR_TASKS == NKT ? 1 : -1];
```