CHAPTER 9

Dynamic Registers

"What a depressingly stupid machine," said Marvin and trudged away.
—Douglas Adams, *The Restaurant at the End of the Universe*

IN ADDITION TO THE AUTOMATIC, or in exceptional cases global, CLINT objects used up to now, it is sometimes practical to be able to create and purge CLINT variables automatically. To this end we shall create several functions that will enable us to generate, use, clear, and remove a set of CLINT objects, the so-called register bank, as a dynamically allocated data structure, where we take up the sketch presented in [Skal] and work out the details for its use with CLINT objects.

We shall divide the functions into private management functions and public functions; the latter of these will be made available to other external functions for manipulating the registers. However, the FLINT/C functions do not use the registers themselves, so that complete control over the use of the registers can be guaranteed to the user's functions.

The number of registers available should be configurable while the program is running, for which we need a static variable NoofRegs that takes the number of registers, which is predefined in the constant NOOFREGS.

```
static USHORT NoofRegs = NOOFREGS;
```

Now we define the central data structure for managing the register bank:

```
struct clint_registers
{
  int noofregs;
  int created;
  clint **reg_l;     /* pointer to vector of CLINT addresses */
};
```

The structure clint_registers contains the variable noofregs, which specifies the number of registers contained in our register bank, and the variable created, which will indicate whether the set of registers is allocated, as well as the pointer reg_l to a vector that takes the start address of the individual registers:

```
static struct clint_registers registers = {0, 0, 0};
```

Now come the private management functions allocate_reg_l() to set up the register bank and destroy_reg_l() to clear it. After space for the storage of the addresses of the registers to be allocated has been created and a pointer is then set to the variable registers.reg_l, there follows the allocation of memory for each individual register by a call to malloc() from the C standard library. The fact that CLINT registers are memory units allocated by means of malloc() plays an important role in testing the FLINT/C functions. We shall see in Section 13.2 how this makes possible the examination of any memory errors that may occur.

```
static int
allocate_reg_l (void)
{
  USHORT i, j;
```

First, memory is allocated for the vector of register addresses.

```
  if ((registers.reg_l = (clint **) malloc (sizeof(clint *) * NoofRegs)) == NULL)
    {
      return E_CLINT_MAL;
    }
```

Now comes the allocation of individual registers. If in the process a call to malloc() ends in an error, all previously allocated registers are cleared and the error code E_CLINT_MAL is returned.

```
  for (i = 0; i < NoofRegs; i++)
    {
      if ((registers.reg_l[i] = (clint *) malloc (CLINTMAXBYTE)) == NULL)
        {
          for (j = 0; j < i; j++)
            {
              free (registers.reg_l[j]);
            }
          return E_CLINT_MAL;     /* error: malloc */
        }
    }
  return E_CLINT_OK;
}
```

The function destroy_reg_l() is essentially the inverse of the function create_reg_l(): First, the content of the registers is cleared by overwriting them