# DS80C400 Silicon Software Programmer's Guide



## 1 Introduction

This document describes the features, components and implementation details of the DS80C400 Silicon Software. A programming reference and sample code provide the necessary equipment to develop network-enabled applications for embedded devices.

### 1.1 Features

The DS80C400 Silicon Software provides the means to load application code

- via the serial port and
- over the network using the NetBoot feature.

In addition to loading code, the DS80C400 Silicon Software contains

- a full TCP/IP IPv4 and IPv6 stack with a Berkeley and industry standard compatible socket interface and
- a preemptive task scheduler which supports multiprocessing and multi-tasking.

The DS80C400 Silicon Software contains everything an application designer needs to network enable his device with minimal effort. What's more, all the built-in features are designed in such a way that they can be easily, selectively enhanced or replaced by the user.

The DS80C400 Silicon Software supports a choice of programming languages, including

- Java (using the TINI OS runtime),
- assembly language and
- C.

The DS80C400 Silicon Software is extremely resource friendly. Only 64 KB of external data memory are required for a fully functioning network server! Because all of the functionality is packed into the DS80C400 Silicon Software, the footprint of user applications can be very small.

The unique, reliable DS80C400 Silicon Software network stack supports both IP version 4 *and* its successor, next-generation IP version 6, ensuring that devices and applications developed today will not be outdated in a year. In addition, the DS80C400 Silicon Software also supports IP version 4 multicasting and a host of other protocols:

- IPv4 and IPv6
- TCP
- UDP
- IGMP for IPv4 multicasting
- ICMP
- DHCP for IPv4
- IPv6 autoconfiguration
- TFTP
- ARP and NDP

Using these protocols, the user can implement network clients and servers running concurrently on the same device.

Table 1 shows the building blocks of the DS80C400 Silicon Software:

Table 1. **DS80C400 Silicon Software Components**

| |
|---|
| Loader |
| Serial Loader / Serial I/O |
| Small Memory Manager, Utility Functions |
| Task Scheduler |
| Ethernet Driver |
| TCP/IP v4/v6 and Socket Layer |
| DHCP |
| TFTP |
| NetBoot |
| Export Table |

These blocks will be described in full detail later in this document.

## 1.2 Hardware Requirements

The following is a summary of the hardware requirements for the DS80C400 Silicon Software:

- DS80C400 microprocessor
- 64 KB of SRAM memory[1]
- Memory (SRAM or flash) to store user application code
- DS2502U-E48 1-Wire chip with MAC address
- External crystal for processor operation

NetBoot will function if the crystal frequency is at least 7 MHz. The network timers are derived from the DS2502 1-Wire bit timing and are accurate to within ± 50% which is sufficient for TCP/IP networking.



1. Must be mapped at address `000000`.

The serial loader supports auto baud to any baud rate, provided the result of the following equation is greater than or equal to one and the differrence to an integer number is within ± 2.5% of its value:

$$r = \frac{\text{Osc. Frequency}}{32 * \text{Baud Rate}}$$

For example, at 18.432 MHz, 115,200 baud are supported, since

$r = \frac{18432000}{32*115200} = 5$.

The functionality was designed to work with crystal oscillators from 3.680 MHz to 75.000 MHz and baud rates from 2400 bps to 115200 bps.

# 2 Boot Process

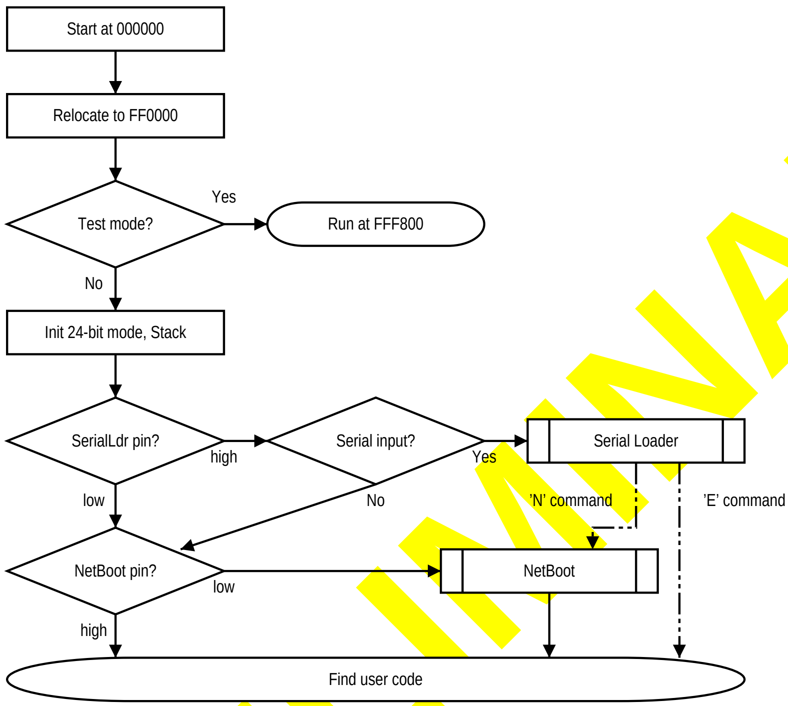
The DS80C400 Silicon Software supports two ways of loading application code. Code can be loaded in traditional manner over the serial port (using an interactive loader) and stored in non-volatile memory (e.g. flash or battery backed SRAM), or it can be loaded over the network in fully automated fashion—since the NetBoot process is so fast, it is feasible to reload the code every time the system boots and store it in volatile memory. If code is stored in non-volatile memory, NetBoot can optionally verify the application code against a centrally stored image on every boot and update the application code automatically if there is any difference.

The boot loader also comes with a mechanism to detect the start address of user applications, leaving the choice of the memory layout in the hands of the user.

## 2.1 Overview

The following chart (Figure 1) shows an overview of the boot process. The interactive serial loader, NetBoot and the feature to locate user applications are fully described in Section 2.2, Section 2.3 and Section 2.4.

Figure 1. **Boot Process**






If the DS80C400 Silicon Software is enabled, the DS80C400 begins execution at memory location `000000` (see data sheet). First, the lower part of the DS80C400 Silicon Software is relocated to `FF0000` and execution continues with initialization of the memory mode (24-bit contiguous mode), stack (1024 Bytes extended stack), and other machine configuration registers (after the relocation of the DS80C400 Silicon Software, the code determines whether the controller is in test mode; if it is in test mode, execution control is transferred to the test routines located at within the ROM).

Next, the code determines what type of reset was executed. In the event of a "clean" reset (i.e. not power failure), it checks the $\overline{\text{SerialLoader}}$ pin (P1.7). If this pin is open (logic high), the code determines whether a carriage return (CR) chracter is transmitted over the serial interface 0 and matches the baud rate. If the CR was received in time, the user is presented with the Serial Loader, which provides options to erase/load flash, dump/execute memory, etc. The JavaKit terminal program supplied by Dallas Semiconductor has a "reset" function that executes the correct reset sequence and triggers the baud rate detection.

If no serial activity is detected and NetBoot is enabled (port pin P5.3 closed or logic low), the DS80C400 Silicon Software continues with NetBoot. Otherwise, execution is transferred to the user code. If the $\overline{\text{SerialLoader}}$ pin is closed (logic low), the DS80C400 Silicon Software will not interact with the serial port at all (no status messages and no serial loader operations).

## 2.2 The Serial Loader

The interactive serial loader displays a copyright notice and a command prompt. Several commands support an optional 'range' parameter. This parameter is interpreted as 'start offset' 'length', e.g. 1000 200 is the range from 1000 to 1200.

The serial loader manages memory in 64 KB blocks ('banks'). A bank (also 'most significant byte') is the high 8 bits of a 24-bit memory address. Most commands apply to the selected bank. Table 2 shows all supported serial loader commands.

Table 2. **Serial Loader Commands**

| **Command** | **Explanation** |
|---|---|
| `B bank` | Selects a bank. Example: `B C0` |
| `C [range]` | Calculates the CRC (with optional range) in the selected bank. *Example:* `C 1000 200` |
| `D [range]` | Dump memory in selected bank in hex format. *Example:* `D 0 20` |
| `E` | Exit the loader and try to execute user code. |
| `F value [range]` | Fill range in selected bank with a byte value. *Example:* `F 00` |
| `G` | 'Goto'—Start executing the selected bank at offset 0. |
| `H, ?` | Help'—Display version number and current bank. |
| `L` | Load hex |
| `N` | Start NetBoot |
| `T [arguments]` | Reserved by Dallas Semiconductor for Test commands |
| `V` | Verify hex |

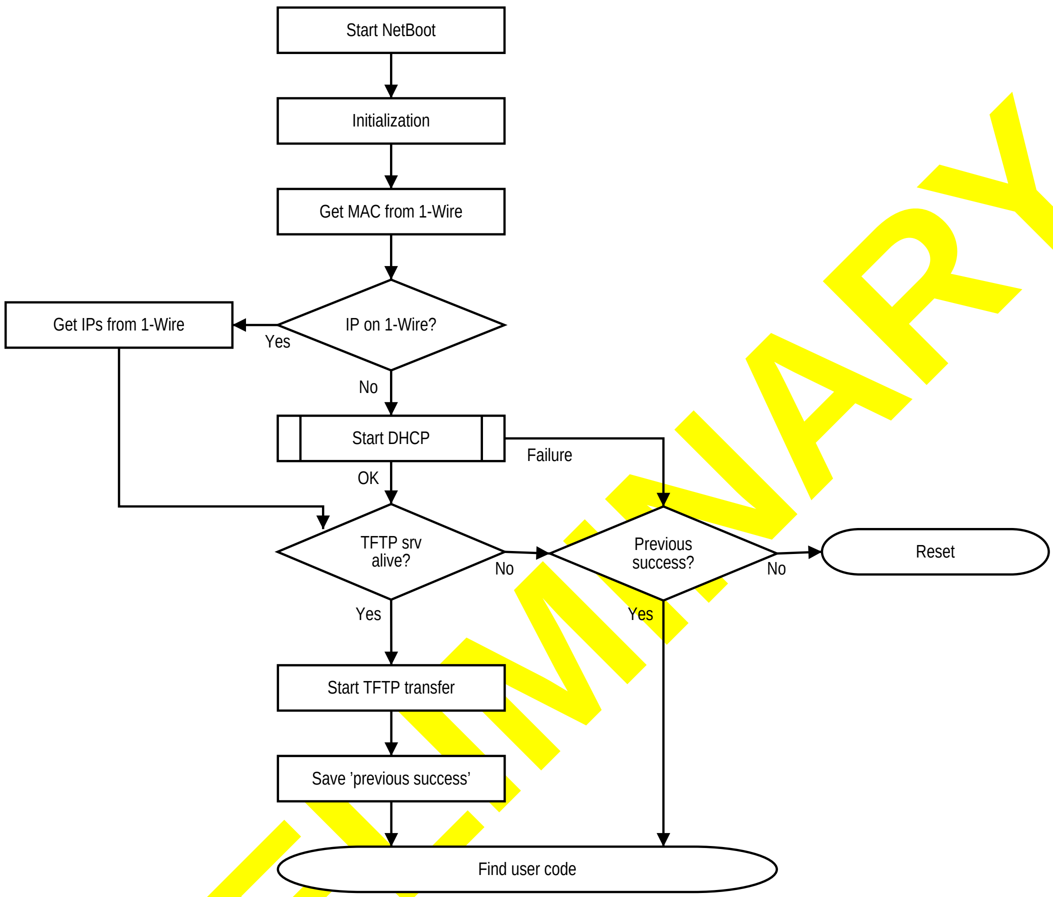
Table 2. **Serial Loader Commands**

| **Command** | **Explanation** |
|---|---|
| `X [offset]` | Execute code at the given offset in current bank. |
| `Z bank` | Zap (erase) flash bank. *Example:* `Z C0` |

## 2.3 NetBoot

The NetBoot feature loads user application code over the network. The code can be loaded on every boot or on demand from the interactive serial loader. NetBoot automatically verifies the code to be loaded against the previously loaded image and skips unnecessary loading steps. This feature both helps to prevent premature aging of flash memory and also ensures that the application code is always current. The following chart (Figure 2) shows the NetBoot process in more detail.

Figure 2. **NetBoot**






After set-up of the interrupt vectors (Ethernet, timer), the DS80C400 Silicon Software memory manager[1] and support functions are initialized. Then, the Ethernet driver, TCP/IP stack and socket layer are initialized. The DS2502U-E48 1-Wire chip with MAC address is required for successful initialization.

1. The DS80C400 Silicon Software memory manager is a minimalistic memory manager which is used mainly during NetBoot. For user application purposes, it can be fully replaced, see Section 4.

Note: Even though NetBoot uses memory at address `000000`, it does not touch one specific 4 KB block. This block is under full control of the user and will neither be read nor written by any portion of the DS80C400 Silicon Software. The memory block is referred to as 'block of binary data' or 'BLOB' in this documentation[1].

### 2.3.1 DHCP

After these initial steps, the code tries to acquire an IP address and the address of a TFTP server using the Dynamic Host Configuration Protocol (DHCP). Specifically, the "next server IP" field of the DHCP acknowledgment packet is used to determine the TFTP server IP. If the site-specific (user defined) option 150 is present in the DHCP packet, it overrides the next server IP (option 150 is also used on Cisco IP phones to get a TFTP server IP address, but is not standardized by an RFC; see Appendix D for details).

### 2.3.2 TFTP and the *tbin2* File Format

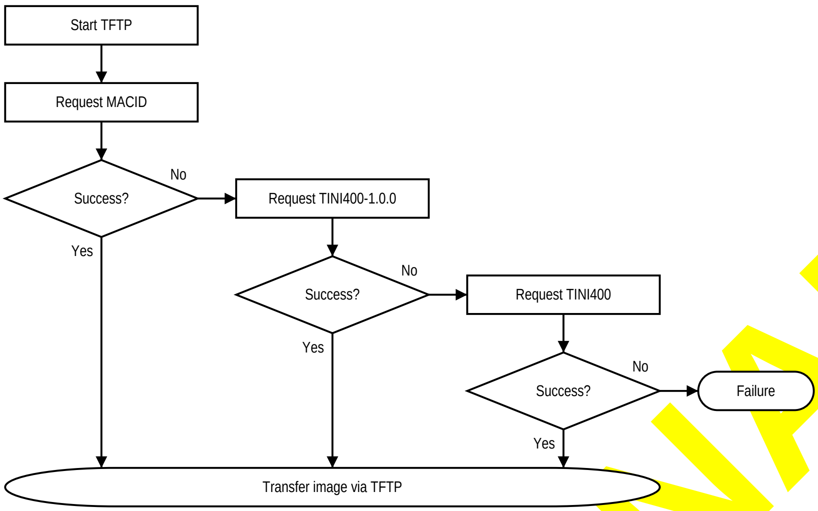
NetBoot first tries to transfer the file 'MACADDRESS' (e.g. `006035AB9811`), then the file 'TINI400-version' (e.g. `TINI400-1.0.1`[2]), and, if this fails, the file `TINI400` (see Table 3). This allows the TFTP server operator to distinguish between different devices and/or different releases of the DS80C400 Silicon Software. The transferred file is loaded into user code space. Subsequently, the DHCP IP address is released and execution control is transferred to the user code.

If NetBoot is unable to receive a response from the TFTP server, it checks whether the previous TFTP transfer completed successfully. If so, NetBoot executes the code. If no, the device is reset. Thus, the TFTP server need not be operational all the time as long as one transfer previously completed successfully. The state information is stored as part of the function redirect table at memory location `000100`; if this memory is not non-volatized, the state information is lost on power-on-reset.

---

1. See Appendix A how to determine the correct address of the block.
2. See Section 9 to find out the version of the DS80C400 Silicon Software.

Table 3. **TFTP File Name Selection**




The NetBoot code uses the Dallas *tbin2* format as its native binary format for loading flash memory. Currently, both read-write RAM (e.g. SRAM) and AMD compatible flash chips are supported.

The *tbin2* format consists of one or more records (see Table 4). The format allows binary concatenation of multiple images into one file, for example the TINI runtime and the TINI user shell slush.

Table 4. **tbin2 Record**

| Field Name | Size | Contents |
|---|---|---|
| Version | 1 | 1 |
| Start address | 3 | Target address for data block (LSB first) |
| Length-1 | 2 | Length-1 of data block (LSB first) |
| CRC-16 | 2 | CRC-16 of data block (LSB first) |
| (Data) | * Length | Binary data |

Note: Versions other than 1 and target addresses above `FF0000` are currently reserved and will be used to support user definable flash programming and memory loading procedures in the future.

Before erasing and loading, the NetBoot TFTP process checks the CRC-16 of the current memory contents. If these contents match, the data block is ignored. Otherwise, whenever the high 8 bits of the start address change, the NetBoot TFTP process determines whether the memory block referenced is SRAM or flash. If the repeated write/read-back of different bit patterns fails, the code concludes the memory must be flash. It then erases the whole 64 KB bank of memory.

Next, NetBoot either programs the binary data into the flash or just writes the data to SRAM, depending on the memory type. The code then calculates the CRC-16 of the block and compares it to the CRC-16 in the *tbin2* description block. If the CRC-16 doesn't match, the microcontroller is reset.

**Note:** The DS80C400 revision B1 requires the segments and checksums in the *tbin2* file to be in a special order. Please see the revision notes.

### 2.3.3 Using 1-Wire instead of DHCP

Instead of using DHCP, the IPv4 address configuration and TFTP server IP can be stored in any of the 1-Wire chips or iButtons listed in Appendix C. Note that the optional part holding the IPv4 address configuration is different from the mandatory part used for the MAC ID. If only a TFTP server IP is stored on 1-Wire, DHCP is still used to acquire the IP address (this allows for remote boot across administrative boundaries).

### 2.3.4 Using the Copyright Message as Progress Indicator

The copyright message is printed in several stages during NetBoot. Each word signifies the successful initialization of a discreet portion of the boot process (see Table 5) and can therefore be used to diagnose configuration problems.

Table 5. **Progress Indication by Copyright Message**

| Display String | Successful initialization of... |
|---|---|
| `... Maxim Integrated` | Interrupt vector table and memory |
| `S/N:` | 1-Wire and Network |
| (Serial number) | Ethernet driver |

Table 5. **Progress Indication by Copyright Message**

| **Display String** | **Successful initialization of...** |
|---|---|
| `MAC ID:` | Sockets |
| (MAC ID) | System timer |

The complete copyright message is similar to the following:

```
DS80C400 Silicon Software - Copyright (C) 2002 Maxim Integrated Products.
S/N: NNNNNNNNNNNNNNNN MAC ID: MMMMMMMMMMMM
```

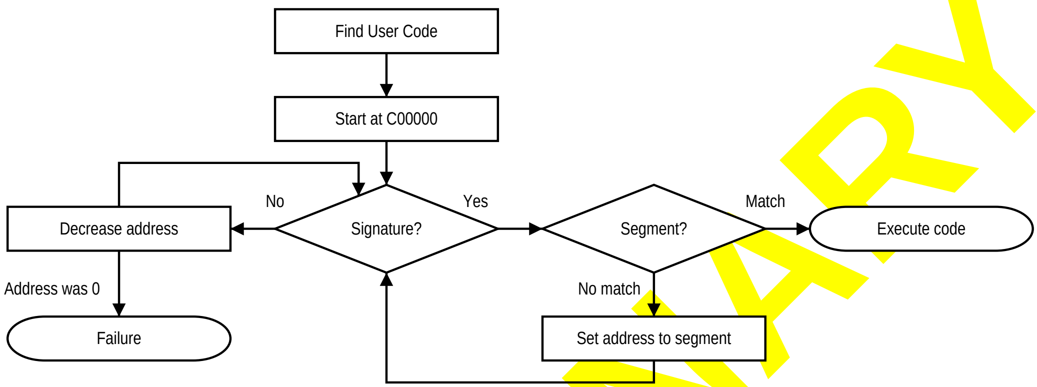
## 2.4 User Code

The DS80C400 Silicon Software searches code space from `C00000` downwards in 64 KB multiples for user code (Figure 3). If it detects the signature—'TINI'—at relative offset 2 (see Table 6), it examines the following byte. This byte contains either 0, in which case the DS80C400 Silicon Software immediately transfers control, or a different value, in which case the DS80C400 Silicon Software compares this byte to the high 8 bits of the current memory address, transferring execution only if they match. This scheme supports configurations where a single memory might appear several times in the code space[1]. If the addresses do not match, the DS80C400 Silicon Software continues the search at 'Mem Addr'`0000`.



1. This might happen when not fully decoding all memory address lines (saving logic chips).

Figure 3. **Determining the Start Address of User Code**




The user code is executed from offset 0 (`sjmp`). Note: The DS80C400 Silicon Software code uses addressing mode dependent instructions only after programming the `acon` register. The user code should do the same (hence `sjmp`, not `ajmp` or `ljmp`).

Table 6. **Signature for User Code**

| Offset | Length in bytes | Contents | Example |
|---|---|---|---|
| 0 | 2 | sjmp xxx | sjmp 07h (ignored in comparison) |
| 2 | 4 | TINI | TINI |
| 6 | 1 | Mem Addr | 0c0h |

# 3 Memory Layout

To use the NetBoot functionality, a minimum of 64 KB fast memory is required at address `000000` (the memory must be fast because the interrupt vector table is stored at address `000000` and executed as code, and this type of access cannot be stretched). Even if there is more memory, the DS80C400 Silicon Software code only uses 64 KB.

The DS80C400 Silicon Software code initializes the memory to the merged *program=data* mode, leaving the $\overline{\mathrm{PCE}x}$ pins as general port pins for the user[1]. The complete memory map after initialization is shown in the following picture:



1. If the user chooses to implement his own memory layout, he must ensure that $\overline{\mathrm{CE0}}$ is set to merged *program=data*, because the first 64 KB of memory is used as interrupt vector table and network stack data storage.

Table 7. **Typical Memory Map**

| **CODE Address** | **Description** | **DATA Address** |
|---|---|---|
| 000000 | $\overline{CE0}$ program=data (minimum of 64 KB required) | 000000 |
| 200000 | $\overline{CE1}$ (data/code) | 200000 |
| 400000 | $\overline{CE2}$ (data/code) | 400000 |
| 600000 | $\overline{CE3}$ (data/code) | 600000 |
| 800000 | $\overline{CE4}$ (data/code) | 800000 |
| A00000 | $\overline{CE5}$ (data/code) | A00000 |
| C00000 | $\overline{CE6}$ (data/code) | C00000 |
| E00000 | $\overline{CE7}$ (recommended I/O space) | E00000 |
| FF0000 | DS80C400 Silicon Software | |
| | SRAM (CAN/Ethernet buffer) | FFDB00 |

Except for the first 64 KB in $\overline{\text{CE0}}$ (interrupt vector table) and the last 64 KB of $\overline{\text{CE7}}$ (DS80C400 Silicon Software), the user can choose a different layout. Different block sizes for $\overline{\text{CE}x}$ (shown in Table 7: 2 MB) are also possible. All $\overline{\text{PCE}x}$ are set to 1 MB and ignored by the NetBoot code. All memory is programmed to *program=data* mode. The recommended I/O space is $\overline{\text{CE7}}$. If stretch cycles are used for the RAM, the first 256 bytes must be shadowed by SRAM because of the interrupt vector table at address `000000`[1].

**Use of the first 64 KB of memory.** While ignoring the rest of the memory, NetBoot makes use of the first 64 KB of RAM for the interrupt vector table, function redirect table, data structures and network buffers. The layout of this area is as follows:

Table 8. **First 64 KB Memory Usage**

| Start address | Description | Write access by |
|---|---|---|
| 000000 | Interrupt vector table (256 bytes) | NetBoot |
| 000100 | Function redirect table (128 bytes) | NetBoot |
| 000180 | Network and task manager data structures | Sockets, NetBoot |
| BLOB | Ignored by DS80C400 Silicon Software ('BLOB') | — |
| BLOB+4K | Network stack buffers and heap | NetBoot, Sockets[1] |
| 010000 | End of memory used by the DS80C400 Silicon Software Environment | — |

1. Only when using the DS80C400 Silicon Software memory manager. Otherwise usage depends on the memory manager.

Note that all memory between `000000` and `BLOB`[2], as well as `BLOB+4K` and `010000` will be erased every time NetBoot functionality is requested. The `BLOB` area is ignored by all DS80C400 Silicon Software code. The socket interface does not write to the block of memory between `000100` and `000180`.



1. If stretch cycles are used, native libraries in the TINI runtime environment cannot be loaded to RAM.
2. See Appendix A to determine the actual address.

# 4 Function Redirect Table—Replacing Built-In Functions

Since the socket interface is used by both NetBoot (from DS80C400 Silicon Software) as well as user code (possibly running under a runtime environment or operating system), the code must be flexible enough to support all types of memory managers, as well as task and thread schedulers. Therefore, the DS80C400 Silicon Software socket interface code does not call these functions directly, but it makes use of a function redirect table. During a NetBoot, the DS80C400 Silicon Software provides its own minimal implementations of these functions. However, to use the socket layer from an application, the user can substitute her own implementations for the following functions:

Table 9. **Function Redirect Table**

| Name | Table offset | Description | Group |
|---|---|---|---|
| BootState | 00h | (used by NetBoot) | — |
| KernelMalloc | 03h | Allocates (fast) kernel memory | Memory Manager |
| KernelFree | 06h | Frees kernel memory | |
| Malloc | 09h | Allocates and clears memory | |
| Free | 0ch | Frees memory | |
| MallocDirty | 0fh | Allocates memory without clearing it | |
| Deref | 12h | Dereferences memory pointer | |
| GetFreeRAM | 15h | Returns free memory | |
| GetTimeMillis | 18h | Returns uptime in milliseconds | Task Manager |
| GetThreadID | 1bh | Returns thread ID | |
| ThreadResume | 1eh | Resumes thread | |
| ThreadIOSleep | 21h | Sleeps, waiting for I/O | |
| ThreadIOSleepNC | 24h | Sleeps, waiting for I/O (run from critical section) | |
| ThreadSave | 27h | Saves thread | |
| ThreadRestore | 2ah | Restores thread | |
| Sleep | 2dh | Sleeps for a number of milliseconds | |
| GetTaskID | 30h | Returns task ID (PID) | |

Table 9. **Function Redirect Table**

| Name | Table offset | Description | Group |
|---|---|---|---|
| InfoSendChar | 33h | Prints debug character to debug port | — |
| IPChecksum | 36h | Computes IP Checksum | — |
| (unused) | 39h | (currently unused) | — |
| DHCPNotify | 3ch | (Hook) Notification of DHCP state change | — |
| TaskCreate | 3fh | (Hook) Primordial task creation | DS80C400 Silicon Software Task Manager Extension[1] |
| TaskDuplicate | 42h | (Hook) Duplication of task | |
| TaskDestroy | 45h | (Hook) Destroying a task | |
| TaskSwitchIn | 48h | Switches current task out | |
| TaskSwitchOut | 4bh | Switches task in | |
| GetMACID | 4eh | Reads MAC ID from DS2502 1-Wire device and IP / Gateway / TFTP Server from other 1-Wire device | — |
| (reserved) | 51h | (reserved) | — |
| Underef | 54h | Reverse of Deref | Mem. Mgr. |
| UserIOPoll | 57h | Called by scheduler | — |
| ErrorNotification | 5ah | Called when out of memory &al. | — |

1. Note: It is possible to either extend the DS80C400 Silicon Software task manager (see Section 6) or to completely replace it.

Functions should be replaced in groups, e.g. if the user provides his own memory manager, all the memory manager functions should be replaced.

Note: All DS80C400 Silicon Software functions (including the exported functions) make heavy use of this table. It must therefore always exist, either in its default state or modified by the user. The function redirect table that is contained in the DS80C400 Silicon Software itself is copied to memory using the `ROM_Redirect_Init` function. NetBoot calls this function. If the user does not use NetBoot, she must call `ROM_Redirect_Init` herself. `ROM_Redirect_Init` restores the function redirect table without altering any other state.

## KernelMalloc

*Allocates fast kernel memory.*

This function allocates a block from the kernel memory pool without incurring the overhead of the regular memory manager.

Table 10. **KernelMalloc register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| r3:r2 | Requested block size | a | 0 if successful |
| | | r3:r2 | Handle of memory block |
| | | dptr0 | Pointer to memory block |

The DS80C400 Silicon Software version of this function is exported as `rom_kernelmalloc`.

## KernelFree

*Frees fast kernel memory.*

This function frees a block of memory allocated by `KernelMalloc`.

Table 11. **KernelFree register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| r3:r2 | Handle of memory block | a | 0 if successful |

The DS80C400 Silicon Software version of this function is exported as `rom_kernelfree`.

## Malloc

*Allocates memory and clears it.*

This function allocates memory from the heap and clears it.

Table 12. **Malloc register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| r3:r2 | Requested block size | a | 0 if successful |
| | | r3:r2 | Handle of memory block |
| | | dptr0 | Pointer to memory block |

The DS80C400 Silicon Software version of this function is exported as `rom_malloc`.

## Free

*Frees memory.*

This function frees a block of memory allocated by `Malloc` or `MallocDirty`.

Table 13. **Free register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| r3:r2 | Handle of memory block | a | 0 if successful |

The DS80C400 Silicon Software version of this function is exported as `rom_free`.

## MallocDirty

*Allocates memory without clearing it.*

This function allocates memory from the heap without clearing it.

Table 14. **MallocDirty register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| r3:r2 | Requested block size | a | 0 if successful |
| | | r3:r2 | Handle of memory block |
| | | dptr0 | Pointer to memory block |

The DS80C400 Silicon Software version of this function is exported as `rom_malloc_dirty`.

## Deref

*Dereferences a memory handle.*

This function dereferences a handle into an absolute address.

Table 15. **Deref register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| r3:r2 | Handle of memory block | a | 0 if successful |
| | | dptr0 | Pointer to memory block |

The DS80C400 Silicon Software version of this function is exported as `rom_deref`.

## GetFreeRAM

*Gets the amount of free memory in the heap.*

This function returns the amount of memory that is still available in the heap.

Table 16. **GetFreeRAM register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | a | 0 if successful |
| | | r3:r0 | Number of free bytes |

The DS80C400 Silicon Software version of this function is exported as `rom_getfreeram`.

## GetTimeMillis

*Gets the current time in milliseconds.*

The DS80C400 Silicon Software does not support a real-time clock, the DS80C400 Silicon Software version of this function therefore returns the number of milliseconds since the system was initialized. A user replacement of this function should return the absolute time.

Table 17. **GetTimeMillis register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | r4:r0 | Time in milliseconds |

## GetThreadID

*Gets the current thread ID.*

Table 18. **GetThreadID register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | a | Current thread ID |

The DS80C400 Silicon Software does not support threads, the DS80C400 Silicon Software version of this function therefore always returns 1.

## ThreadResume

*Resumes a suspended thread.*

Table 19. **ThreadResume register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | Thread ID | a | 0 if successful |
| r0 | Task ID | | |

The DS80C400 Silicon Software does not support threads. The DS80C400 Silicon Software version of this function therefore resumes the task.

## ThreadIOSleep
## ThreadIOSleepNC

*Puts a thread to sleep, waiting for I/O.*

The `NC` version of this function does not enter a critical section and is only called from critical sections.

Table 20. **ThreadIOSleep register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | Non-zero: Infinite timeout | a | 0 if I/O, else timeout |
| `r3:r0` | Timeout value (if a is zero) | | |

The DS80C400 Silicon Software does not support threads. The DS80C400 Silicon Software version of this function therefore works on the task.

## ThreadSave
## ThreadRestore

*Saves / restores a thread.*

This function saves / restores the state of a thread.

Table 21. **ThreadSave / ThreadRestore register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | — | |

The DS80C400 Silicon Software does not support threads. The DS80C400 Silicon Software versions of these functions therefore do nothing.

## Sleep

*Sleeps.*

This function suspends a task for *at least* the requested amount of time.

Table 22. **Sleep register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | Task ID | a | 0 if successful |
| `r3:r0` | Sleep time in milliseconds | | |

## GetTaskID

*Gets the current task ID.*

Table 23. **GetTaskID register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | a | Task ID |

The DS80C400 Silicon Software version of this function is exported as `task_getcurrent`.

## InfoSendChar

*Sends a character to the serial port.*

Table 24. **InfoSendChar register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | Character to send | — | |

The DS80C400 Silicon Software version of this function accesses the $\overline{SerialLdr}$ pin (see Section 2.1) and does nothing if this pin is low. The DS80C400 Silicon Software does not use interrupt driven I/O to the serial port.

## IPChecksum

*Calculates the IP checksum for an IP packet.*

Table 25. **IPChecksum register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| r3:r0 | Initial checksum | r1:r0 | Checksum |
| r5:r4 | Size of buffer | | |
| dptr0 | Pointer to buffer | | |

The DS80C400 Silicon Software version of this function does not use the checksum accelerator feature of the DS80C400.

## DHCPNotify

*Notifies of a DHCP state change.*

Table 26. **DHCPNotify register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | New DHCP state | — | |

The DS80C400 Silicon Software version of this function is exported as `rom_dhcp_notify` (see `dhcp_init()` and `dhcp_status()` for a complete description).

## TaskCreate
## TaskDuplicate
## TaskDestroy
## TaskSwitchIn
## TaskSwitchOut

*Task scheduler functions.*

These hooks are documented in Section 6.

## GetMACID

*Reads the MAC ID.*

This function reads the MAC ID and stores it in the `MAC_ID` variable.

Table 27. **GetMACID register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | a | 0 if successful |

The DS80C400 Silicon Software version of this function accesses the 1-Wire port and searches for a DS2502U-E48 1-Wire chip.

## Underef

*Resolves a physical address into a memory handle.*

This function un-dereferences an absolute address and returns a memory handle.

Table 28. **Underef register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| `dptr0` | Pointer to memory block | a | (destroyed) |
| | | `r3:r2` | Handle of memory block |

See also `Deref`.

## UserIOPoll

*Allows extension of the IOPoll mechanism.*

This function is called on every `IOPoll`, i.e. driven by the timer when no interrupts are active. A user could add housekeeping functions, for example.

Table 29. **UserIOPoll register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | — | |

The default implementation of this function does nothing.

## ErrorNotification

*Notifies of an error.*

This function is called when the system doesn't have enough memory and other error situations. A user implementation could print an error message or reset the system.

Table 30. **ErrorNotification register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | Error number | — | |

The default implementation does nothing. The following error codes are currently defined:

Table 31. **Error codes**

| Symbol | Value | Description |
|---|---|---|
| ERROR_KMEM | 0 | Kernel memory exhausted |
| ERROR_MRM | 1 | Heap memory exhausted |
| ERROR_TCP_MEM | 2 | TCP detected low memory |
| ERROR_TCP_RESEND | 3 | TCP packet has to be resent |
| ERROR_KFREE_FAIL | 4 | Attempt to free a kernel memory block failed |

Table 31. **Error codes**

| Symbol | Value | Description |
|---|---|---|
| ERROR_FREE_NULL | 5 | Attempt to free a NULL pointer |
| ERROR_FREE_DEREF | 6 | Could not dereference a memory handle on free |

# 5 TCP/IP and the Socket Layer

## 5.1 Overview

The DS80C400 Silicon Software supports TCP/IP Ethernet networking over Berkeley/industry standard “socket” functions, which are widely used across all operating systems. Making use of the networking functionality or porting existing code to the DS80C400 Silicon Software Environment should therefore be an easy task for most developers.

## 5.2 Features

- TCP and UDP client and server sockets on IPv4 and IPv6
- Multicasting (IPv4 only)
- ICMP echo request (“ping”) for IPv4 and IPv6
- DHCP client (IPv4 only)
- TFTP client for IPv4 and IPv6
- Supports up to 24 sockets

## 5.3 Limitations

The current IPv4 implementation does not support packet fragmentation or reassembly and IP options. The maximum packet size is therefore 576 (minimum IPv4 packet size to 1500 (local Ethernet).

## 5.4 Socket Functions

The following sections describe the socket functions exported by the DS80C400 Silicon Software. Each function signature is given in C notation, along with sample usage and implementation specific notes. Section 5.4.5 shows how to use these functions from assembly language, Section 5.4.6 gives examples for the Keil C compiler.

**Return values.** Negative values denote errors, zero and greater zero mean *no error*.

**The struct sockaddr.** The *sockaddr struct* is a basic data structure. In the DS80C400 Silicon Software Environment, it is defined as follows:

```
struct sockaddr {
    unsigned char sin_addr[16];          /* sin_port and sin_addr - offset 0 */
    unsigned short sin_port;             /* combined are sin_data - offset 16 */
    char sin_family;                     /* ignored (- offset 18) */
};
```

Note that the address family is ignored; to use IPv4, simply set the first 12 bytes of *sin_addr* to 0. The length of this structure is always 18 or greater. The *length* parameter supplied to various functions is *not* checked (i.e. the *sin_family* need not be present, it is a member of the structure for compatibility reasons only).

## 5.4.1 Standard Socket Functions

The following functions describe Berkeley/industry standard socket functions.

### int socket(int domain, int type, int protocol);

*Creates a network socket (a local endpoint) for TCP or UDP communication.*

*Type* can either be `SOCK_STREAM` for TCP sockets or `SOCK_DGRAM` for UDP sockets. The *domain* and *protocol* parameters are ignored. `socket()` returns a socket handle., i.e. an identifier for the new socket.

Table 32. **Type Values for the socket() call**

| Type | Value |
|---|---|
| SOCK_DGRAM | 0 |
| SOCK_STREAM | 1 |

A newly created socket has no specific local address assigned to it. To use it as a server socket, the use of `bind()` is required.

To use a streaming (TCP) socket, the socket must be connected using either `connect()` or `listen()/accept()`.

To destroy/free a socket, use `closesocket()`.

`socket()` and `accept()` are the only functions that return a socket number. All other socket functions require the socket number to be passed to them to access the correct socket.

Example 1. **socket()**

```
int sock = socket(0, SOCK_STREAM, 0);
```

Note: The `socket()` function calls `GetTaskID()` to get the current task ID.

## int closesocket(int s);

*Closes a socket.*

Closes the socket *s* which was created by the `socket()` call and returns a success/failure code.

Example 2. **closesocket()**

```
closesocket(sock);
```

## int sendto(int s, void *buf, int len, int flags, struct sockaddr *addr, int addrlen);

*Sends a UDP datagram to the specified address.*

The target address is specified in the *addr* parameter, *addrlen* is the size of the *addr* structure. The datagram itself is referenced by *buf*, and *len* is the size of the datagram. The *flags* parameter is ignored.

`sendto()` is unable to detect whether the datagram has successfully reached the destination and only returns a failure code on local errors.

Use `bind()` to specify a local port number. Without `bind()`, `sendto()` chooses a random local port.

Example 3. **sendto()**

```
int result;
struct sockaddr target;
int sock = socket(0, SOCK_DGRAM, 0);
char *buffer = malloc(123);

buffer[0] = .........;
....
memzero(target);
target.sin_addr[0] = .....;
....
target.sin_port = 80;
result = sendto(sock, 123, 0, target, sizeof(struct sockaddr));
```

### int recvfrom(int s, void *buf, int len, int flags, struct sockaddr *addr, int *addrlen);

*Receives a UDP datagram.*

`recvfrom()` receives a message on the socket *s*, storing the message in *buf*. *len* is the size of *buf*. If *addr* is not `NULL`, the remote address is filled in. The *flags* parameter is ignored.

`recvfrom()` returns the number of bytes read.

If no data is available on the socket, `recvfrom()` blocks for the amount of time specified with the `setsockopt/SO_TIMEOUT` call.

Note: It is generally required to use the `bind()` call first to assign a local port to the socket.

Example 4. **recvfrom()**

```
struct sockaddr localaddr;
int result;
int sock = socket(0, SOCK_DGRAM, 0);
char *buffer = malloc(123);
memzero(localaddr);
localaddr.sinPort = 7;
bind(sock, localaddr, sizeof(struct sockaddr));
result = recvfrom(sock, 123, 0, NULL, 0);
```

## int connect(int s, struct sockaddr *addr, int addrlen);

*Connects a TCP socket to the specified address.*

`connect()` connects the socket s to the remote address specified by the *addr* structure and returns a success/failure code.

`connect()` may only be used once with each socket.

Example 5. **connect()**

```
int result;
struct sockaddr target;
int sock = socket(0, SOCK_STREAM, 0);
memzero(target);
target.sin_addr[0] = .....;
....
target.sin_port = 80;
result = connect(sock, target, sizeof(struct sockaddr));
```

## int bind(int s, struct sockaddr *addr, int addrlen);

*Binds a socket to the specified address.*

This function assigns a local address and port to the socket s. The combination of IP address and port is often referred to as "name". Binding a socket is necessary for server sockets. For client sockets, use `bind()` if a specific source port is requested.

*s* is the socket to be assigned a local name, *addr* contains the local address (IP and port values).

`bind()` returns a success/failure code.

Example 6. **bind()**

```
struct sockaddr localaddr;
int sock = socket(0, SOCK_DGRAM, 0);
memzero(localaddr);
localaddr.sin_port = 6789;
bind(sock, localaddr, sizeof(struct sockaddr));
```

## int listen(int s, int backlog);

*Listens for connections on the specified socket.*

`listen()` creates a queue of length *backlog*; *backlog* is the maximum number of pending new incoming connections (max. 16).

`listen()` returns a success/failure code.

To move an incoming connection request from the queue to an established state, `accept()` must be called. It is generally required to use `bind()` to assign a local name to a socket before invoking `listen()`.

Example 7. **listen()**

```
int result;
struct sockaddr localaddr;
int sock = socket(0, SOCK_STREAM, 0);
memzero(localaddr);
localaddr.sin_port = 80;
bind(sock, localaddr, sizeof(struct sockaddr));
result = listen(s, 5);
```

## int accept(int s, struct sockaddr *addr, int *addrlen);

*Accepts a TCP connection on the specified socket.*

`accept()` moves the first pending new incoming connection request from the listen queue into the established state. It assigns a new local socket (connection endpoint) to the connection and returns its socket handle.

`accept()` blocks if there are no pending new incoming connection requests.

The socket *s* must first be created by the `socket()` call, bound to an address using `bind()` and a listen queue must be created for it using `listen()`.

If *addr* is not `NULL`, the remote address is filled in.

Example 8. **accept()**

```
int result;
struct sockaddr localaddr;
int sock = socket(0, SOCK_STREAM, 0);
memzero(localaddr);
```

```
localaddr.sin_port = 80;
bind(sock, localaddr, sizeof(struct sockaddr));
listen(s, 5);
result = accept(sock, NULL, 0);
```

## int recv(int s, void *buf, int len, int flags);

*Reads data from a connection-oriented (TCP) socket.*

`recv()` reads up to *len* bytes from the socket *s* (which must be in a connected state) into the buffer *buf*. It returns the number of bytes read. The *flags* parameter is ignored. If there is no data on the socket, `recv()` blocks infinitely unless a socket timeout is set using `setsockopt()`.

Example 9. **recv()**

```
int result;
int sock;
char *buf = malloc(123);
....
result = recv(sock, buf, 123, 0);
```

## int send(int s, void *buf, int len, int flags);

*Writes data to a connection-oriented (TCP) socket.*

`send()` writes *len* bytes from the buffer *buf* to the socket *s*, which must be in a connected state. The *flags* parameter is ignored. `send()` returns a local success/failure code. It does not necessarily detect transmission errors.

Example 10. **send()**

```
int result;
int sock;
char *buf = malloc(123);
buf[0] = ....;
....
result = send(sock, buf, 123, 0);
```

### int getsockopt(int s, int level, int name, void *buf, int *len);
### int setsockopt(int s, int level, int name, void *buf, int len);

*Gets and sets socket options.*

`getsockopt()` and `setsockopt()` get/set various options of a socket *s*. The option to be read/written is specified by the *name* parameter. The *buf* parameter points to the buffer to be filled (get operation) or the option value to be written (set operation). *len* is the size of the buffer and modified to the size of the filled-in data by `getsockopt()`. *buf* may be `NULL` if a socket option needs no data. The *level* parameter is ignored. `getsockopt()/setsockopt()` return a success/failure code.

The following option names are supported:

Table 33. **Socket Options**

| Name | Value | Description |
|---|---|---|
| TCP_NODELAY | 0 | Gets/sets the TCP Nagle parameter |
| SO_LINGER | 1 | (ignored) |
| SO_TIMEOUT | 2 | Gets/sets the socket timeout |
| SO_BINDADDR | 3 | Gets the local socket IP (get operation only) |

Example 11. **setsockopt()**

```
int timeout = 5000;
setsockopt(sock, 0, SO_TIMEOUT, &timeout, sizeof(int));
```

### int getsockname(int s, struct sockaddr *addr, int *addrlen);

*Returns current local address of a socket.*

`getsockname()` returns the local IP and port of the socket *s* and stores it in the *addr* structure.

`getsockname()` returns a success/failure code.

Example 12. **getsockname()**

```
struct sockaddr whataddr;
result = getsockname(sock, whataddr, sizeof(struct sockaddr));
```

### int getpeername(int s, struct sockaddr *addr, int *addrlen);

*Returns the remote address of a connection-oriented (TCP) socket.*

If socket *s* is connected, `getpeername()` stores the remote address into the *addr* structure.

`getpeername()` returns a success/failure code.

Example 13. **getpeername()**

```
struct sockaddr remoteaddr;
result = getpeername(sock, remoteaddr, sizeof(struct sockaddr));
```

## 5.4.2 DS80C400 Silicon Software Extensions

The following functions are extensions only found in the DS80C400 Silicon Software Environment.

### int setnetworkparams(void *parameters);
### int getnetworkparams(void *parameters);

*Sets/gets the IPv4 address and configuration parameters.*

`setnetworkparams()`/`getnetworkparams()` allow the user to set/get the IPv4 portion of the network configuration (note: the IPv6 address is auto configured). To auto configure IPv4, use the DHCP functionality instead of `setnetworkparams()`—the address configured by DHCP can be read using `getnetworkparams()`. Both functions return a success/failure code.

*parameters* is a buffer containing the following data:

Table 34. **Description of the IPv4 Network Configuration**

| Parameter | Offset | Length | Description |
|---|---|---|---|
| (zero) | 0 | 12 | must be 0 |
| IP4ADDR | 12 | 4 | IP address |
| IP4SUBNET | 16 | 4 | Subnet mask |
| IP4PREFIX | 20 | 1 | Number of 1 bits in subnet mask |

Table 34. **Description of the IPv4 Network Configuration**

| Parameter | Offset | Length | Description |
|---|---|---|---|
| (zero) | 21 | 12 | must be 0 |
| IP4GATEWAY | 33 | 4 | IP address of default gateway |

Example 14. **getnetworkparams()**

```
unsigned char config[37]
getnetworkparams(config);
```

## getipv6params(void *parameters);

*Gets the IPv6 address.*

This function returns the IPv6 address of the Ethernet interface.

*parameters* is a buffer containing the following data:

Table 35. **Description of the IPv6 Network Configuration**

| Parameter | Offset | Length | Description |
|---|---|---|---|
| IP6ADDR | 0 | 16 | IP address |
| IP6PREFIX | 16 | 1 | IP prefix length |

Example 15. **getipv6params()**

```
unsigned char config[17]
getipv6params(config);
```

## int getethernetstatus(void);

*Returns the Ethernet status (Link).*

The return value is a bit-wise or of the following flags:

Table 36. **Ethernet Status Bits**

| Name | Value | Description |
|---|---|---|
| ETH_STATUS_LNK | 1 | Ethernet link status |

All other flags are currently reserved.

Example 16. **getethernetstatus()**

```
int linkup = getethernetstatus() & ETH_STATUS_LNK;
```

## int gettftpserver(struct sockaddr *addr, int *addrlen);
## int settftpserver(struct sockaddr *addr, int *addrlen);

*Sets/gets the IP address of the TFTP server.*

`gettftpserver()` stores the address of the TFTP server into the *addr* structure. `settftpserver()` sets the TFTP server address to the value supplied in *addr*.

The `settftpserver()` function must be used if the address of the TFTP server is not acquired via DHCP or 1-Wire. Both functions return a success/failure code.

Example 17. **gettftpserver()**

```
struct sockaddr tftpaddr;
result = gettftpserver(sock, tftpaddr, sizeof(struct sockaddr));
```

## int cleanup(int pid);

*Closes all sockets associated with a process ID.*

This function should be called by the user's task manager whenever a process dies to ensure all associated resources are freed by the socket layer. *pid* is the process ID of the terminated process. `cleanup()` returns a success/failure code.

Note: The DS80C400 Silicon Software task scheduler (see Section 6) does not call this function. The user should call `cleanup()` after each `task_kill()` call.

Example 18. **cleanup()**

```
int process = fork(...);
if (process == 0) return;
...
task_kill(process);
cleanup(process);
```

### int avail(int s);

*Returns bytes available for read on a socket.*

This function reports the number of bytes available for `read()` on a connection-oriented (TCP) socket.

Example 19. **avail()**

```
int numbytes = avail(sock);
```

### int join(int s, struct sockaddr *addr, int addrlen);
### int leave(int s, struct sockaddr *addr, int addrlen);

*Joins/leaves the specified multicast group.*

Use `join()` and `leave()` to add a datagram socket *s* to a multicast group. The group name is specified by the *addr* structure. `join()`/`leave()` return a success/failure code.

Note: The current implementation does not support IPv6 multicasting.

Example 20. **join()**

```
struct sockaddr group;
int sock = socket(0, SOCK_DGRAM, 0);
memzero(group);
group.sin_addr[12] = 224;
group.sin_addr[13] = 0;
group.sin_addr[14] = 0;
group.sin_addr[15] = 7;
result = join(sock, group, sizeof(struct sockaddr));
```

### int ping(struct sockaddr *addr, int *addrlen, int TTL, unsigned char *response);

*Pings the specified address and returns the result.*

`ping()` sends an ICMP echo request ("ping") to a remote host specified by *addr*. The packets sent by `ping()` have the specified time to live (*TTL*). `ping()` returns the response time; the buffer pointed to by *response* gets filled in with the data returned.

Example 21. **ping()**

```
struct sockaddr host;
unsigned char buffer[2048];
memzero(host);
host.sin_addr[...] = ...;
result = ping(host, sizeof(struct sockaddr), 20, buffer);
```

### 5.4.3 Using Sockets

To make use of the built-in TCP/IP socket layer, an application (or runtime environment) must run in 24-bit contiguous mode and take the following steps:

- Initialize the sockets layer (see Appendix B).
- Make sure the asynchronous maintenance functions (see Appendix F) are periodically called from the timer interrupt.

### 5.4.4 Calling Conventions for Socket Functions

The DS80C400 Silicon Software socket functions conform to the TINI Native Library calling conventions (NatLib). These calling conventions are documented separately. Section 5.4.6 shows how to call DS80C400 Silicon Software functions using the Keil C compiler.

**Parameters.** The NatLib calling conventions can be summarized as follows: `R7_B2:R5_B2` point to the parameter buffer. Each parameter is 4 bytes wide (the socket functions neither use long nor double). Parameter 0 would be at offset 0, parameter 1 at offset 4, etc. All other registers can and will be destroyed by the socket functions, i.e. the user must take care to save `dptr0`, `dptr1`, `b`, register banks 0, 1 and 2, `dps` and `psw`.

**Integer representation.** 8-bit, 16-bit and 32-bit values are stored LSB first in the corresponding parameter.

**Pointer representation.** Pointers are stored LSB first in bytes 0, 1 and 2 of the corresponding parameter.

**Array representation.** Arrays are represented by type byte (ignored), array length (LSB/MSB) and the array contents. A pointer to the array header is stored in the corresponding parameter.

**Success/failure code.** Every function signals success by returning `a = 0`.

**Integer return values.** 8-bit, 16-bit and 32-bit values are returned in registers `r3:r0`.

Note: The utility, task, DHCP and TFTP functions are using registers to pass values. They do *not* use the NatLib conventions.

### 5.4.5 Calling Socket Functions from Assembly Language Programs

To call socket functions, the NatLib call interface must be emulated. This can be done by declaring a buffer. Each parameter takes up 4 bytes in the buffer:

Example 22. **NatLib parameter buffer**

```
PARAMBUFFER_0   EQU     PARAMBUFFER
PARAMBUFFER_1   EQU     PARAMBUFFER_0+4
PARAMBUFFER_2   EQU     PARAMBUFFER_1+4
PARAMBUFFER_3   EQU     PARAMBUFFER_2+4
PARAMBUFFER_4   EQU     PARAMBUFFER_3+4
PARAMBUFFER_5   EQU     PARAMBUFFER_4+4
PARAMBUFFER_END EQU     PARAMBUFFER_5+4
```

The DS80C400 Silicon Software exports a five-argument `PARAMBUFFER`, see Appendix A. Note: If more than one task needs to use a parameter buffer, the user must either declare separate buffers per task or protect the DS80C400 Silicon Software buffer from concurrent access. The DHCP task uses the DS80C400 Silicon Software `PARAMBUFFER`.

To call a socket function, this buffer must be loaded with the required arguments and the address of the buffer must be stored in `R7_B2:R5_B2`. The following is an example for the `closesocket()` call:

Example 23. **Calling closesocket() in assembly language**

```
; Input: acc = socket
mov     R7_B2, #(PARAMBUFFER shr 16)
mov     R6_B2, #((PARAMBUFFER shr 8) and 0ffh)
mov     R5_B2, #(PARAMBUFFER and 0ffh)
mov     dptr, #PARAMBUFFER_0
PUTX                            ; Store socket number
inc     dptr
clr     a
PUTX
inc     dptr
PUTX
inc     dptr
PUTX

RNLCALL closesocket             ; closesocket(socket)
```

### 5.4.6 Calling Socket Functions using the Keil C Compiler

Calling socket functions in C (using the Keil C compiler) hides the details of the NatLib call interface and the parameter buffer. Calling a socket function from C is as easy as including the correct header file in your project. The following is an example of a simple TCP exchange:

Example 24. **Simple TCP exchange in C**

```
// socket_handle is an unsigned int, assdress is a struct sockaddr
socket_handle = socket(0, SOCKET_TYPE_STREAM, 0);
printf("Result of socket creation: %u\r\n", socket_handle);
for (i=0;i<18;i++)
    address.sin_addr[i] = 0;
address.sin_addr[12] = 180;
address.sin_addr[13] = 0;
address.sin_addr[14] = 64;
address.sin_addr[15] = 118;
address.sin_port = 33333;
connect(socket_handle, &address, sizeof(struct sockaddr));
// data_to_send points to a string
send(socket_handle, data_to_send, strlen(data_to_send), 0)
//buffer is some storage space for output, length is an unsigned int
length= recv(socket_handle, buffer, 10, 0);
printf("Received number of bytes: %x", length);
```

Please see the appendix for more information on using the Keil C Compiler with the DS80C400 Silicon Software.

### 5.4.7 DHCP Functions

The following functions do *not* use the NatLib calling conventions.

Example 25. **IP Address Acquisition Using DHCP**

```
; Start DHCP client and get IP
ROMCALL  dhcp_init
; Wait for DHCP IP
clr      a                         ; This task
mov      b, #TASK_DHCPSLEEP
ROMCALL  task_suspend
; At this point, we have a valid IP...
```

## int dhcp_init(void);

*Initializes the DHCP client.*

`dhcp_init()` starts a DHCP client task and returns to the caller. To read the address DHCP has configured (only valid if DHCP is in bound, renewing or rebinding state, see `dhcp_status()`), use the socket layer function `getnetworkparams()`.

DHCP is implemented for IPv4 only. The IPv6 portion of the TINI400 network stack uses Neighbor Discovery.

The DHCP client calls the `DHCPNotify` redirect when it acquires or loses an IP. The DS80C400 Silicon Software version of `DHCPNotify` is exported as `rom_dhcp_notify` and sends the `TASK_DHCPSLEEP` signal to the task that originally called `dhcp_init()` to wake that task up.

Table 37. **dhcp_init() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | a | Success/failure code |

**int dhcp_status(void);**

*Returns the DHCP state.*

Table 38. **DHCP States**

| Name | Value |
|---|---|
| INIT | 0 |
| SELECTING | 1 |
| REQUESTING | 2 |
| INITREBOOT | 3 |
| REBOOTING | 4 |
| BOUND | 5 |
| RENEWING | 6 |
| REBINDING | 7 |

See RFC 2131 for a description of these states.

Table 39. **dhcp_status() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | a | DHCP status |

**void dhcp_stop(void);**

*Kills the DHCP client.*

`dhcp_stop()` disables the DHCP functionality and kills the DHCP client task.

Table 40. **dhcp_stop() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | — | |

### 5.4.8 TFTP Functions

The following functions do *not* use the NatLib calling conventions.

## int tftp_init(void);

*Initializes the TFTP client.*

`tftp_init()` sets up the data structures required for the TFTP client, most important the `TFTP_MSG` data buffer.

Table 41. **tftp_init() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | a | Success/failure code |

## int tftp_first(char *fname);

*Requests the first TFTP data block and waits for data.*

`tftp_first()` requests the file name *fname* from the TFTP server and returns the number of bytes read. If this number is less than 512, the transfer has ended. This function also allocates a socket handle. Use `tftp_close()` to free the socket handle.

Table 42. **tftp_first() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| `dptr0` | *fname* | a | Success/failure code |
| | | `r1:r0` | Bytes read return value |

### int tftp_next(int ack_only);

*Acknowledges a TFTP data block and waits for data.*

`tftp_next(0)` returns subsequent data blocks (until the returned length is less than 512). Use `tftp_next(1)` to acknowledge the last data block without waiting for additional data.

Table 43. **tftp_next() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | *ack_only* | a | Success/failure code |
| | | r1:r0 | Bytes read return value |

### int tftp_close(void);

*Closes a tftp socket allocated by tftp_first().*

`tftp_close()` frees the socket handle allocated by `tftp_first()` and should be called when a tftp data transfer has finished (or when a tftp data transfer has failed)..

Table 44. **tftp_next() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | a | Success/failure code |

## 5.4.9 TFTP Client Framework

Since the handling of the TFTP data is different for each application, there is no generic “do_tftp” function. The following code implements the framework for a TFTP client:

Example 26. **Simple TFTP Client**

```
; Try to tftp a boot file
ROMCALL  tftp_init
; Set state variables
; ..........
mov      r5, #TFTP_RETRIES
```

```
tftp_retryfirst:
        mov      dptr, #FILENAME
        ROMCALL  tftp_first
        jz       tftp_loop                          ; OK
        cjne     a, #TFTP_TIMEOUT, tftp_failure
        dec      r5                                 ; Retries left?
        mov      a, r5
        jnz      tftp_retryfirst
tftp_failure:
        ; ERROR MESSAGE HERE
        sjmp     tftp_finished
        ; Get next tftp packet
tftp_loop:
        ; PRINT PROGRESS HERE
        mov      r5, #TFTP_RETRIES
        ; Do something with the data we received
        ; HANDLE TFTP_MSG DATA HERE
        ; Check whether this was the last block
        mov      a, r1
        cjne     a, #2, tftp_last
        mov      a, r0
        cjne     a, #4, tftp_last
tftp_retry:
        ; Received 512 bytes of data -- get next block
        clr      a                                  ; Send ACK and receive new data
        ROMCALL  tftp_next
        jz       tftp_loop                          ; OK, next one
        xrl      a, #0feh                           ; Server resent old block
        jz       tftp_retry
        ; PRINT ERROR NOTIFICATION HERE
        dec      r5                                 ; Retries for this block
        mov      a, r5
        jnz      tftp_retry
        sjmp     tftp_failure
tftp_last:
        ; ACK last block
        mov      a, #1                              ; Send ACK only
        ROMCALL  tftp_next
        ; PRINT SUCCESS HERE
tftp_finished:
        ROMCALL  tftp_close
```

# 6 Task Scheduler

## 6.1 Overview

The DS80C400 Silicon Software provides a task scheduler to simplify the implementation of applications such as Serial-to-Ethernet. The implementation is optimized for size and a small number of tasks. To allow for customization, the task scheduler supports hooks.

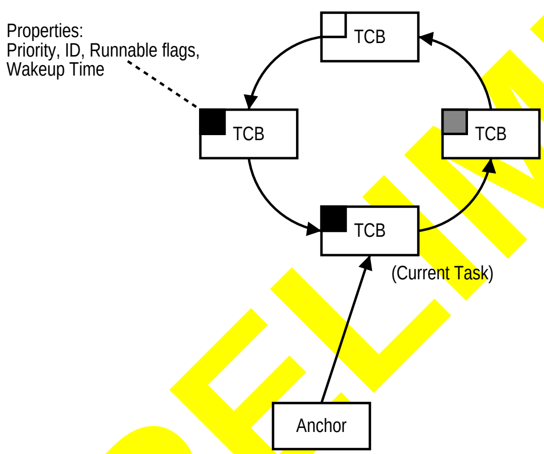
To summarize the features:

- Full task support
- Hooks for threads (the user can supply save/restore state functions)
- Preemptive
- Priority-based, supports an Idle Task

## 6.2 Data Structures

The tasks are organized as a ring of Task Control Blocks (TCBs). Each TCB has attributes such as *Priority* and the *Task ID*. The following picture shows a configuration with four tasks:

Figure 4. **The Task Ring**




- Task Control Block (TCB):

```
Priority   - Priority of this task (8 bits)
             MIN_PRIORITY = 1, NORM_PRIORITY = 80h, MAX_PRIORITY = 0ffh
ID         - ID of the task (8 bits)
             Note: 0 is invalid TaskID
Next       - Pointer to next task in ring
Flags      - 8 bits indicating runnable (all clear) or waiting for an event (bit set)
             Bit 0 = Sleeping (requires timeout)
             Bit 1 = Waiting for IO (timeout allowed)
             Bit 2 = Waiting for DHCP established (timeout allowed)
             Bit 3..7 = Available for user code
WakeupTime - Time value (5 bytes) when task should wake up
StateSize  - Size of the state buffer (16 bits)
StatePtr   - SFRs / stack / PC
```

- Task ring: Linked ring of TCBs (forward pointers only)

- Anchor (Task pointer): Points to running task, also used as an entry to the task ring

Every task can wait for events to happen. These events are stored in the *flags* field of the TCB and defined as follows:

Table 45. **Event Bit Masks**

| Name | Value | Description |
|---|---|---|
| TASK_SLEEPING | 1 | Task is waiting for sleep() to finish |
| TASK_IOSLEEP | 2 | Task is waiting for I/O |
| TASK_DHCPSLEEP | 4 | Task is waiting for 'DHCP established' |
| TASK_USER0 | 8 | User defined |
| TASK_USER1 | 10h | User defined |
| TASK_USER2 | 20h | User defined |
| TASK_USER3 | 40h | User defined |
| TASK_USER4 | 80h | User defined |

## 6.3 Description of Functions

The following functions do not use the NatLib calling conventions.

### void task_genesis(int savesize);

*Sets up primordial and idle threads.*

Creates the running task list and assigns the task ID 1 to the current flow of execution. The function calls `mmalloc()` to allocate a buffer of *savesize*, which is used to save and restore the task's state.

Note: This function does not change or enable the timer interrupt.

Table 46. **task_genesis() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| `r1:r0` | *savesize* | — | |

Example 27. **task_genesis()**

```
mov     r1, #(ROM_SAVESIZE shr 8)
mov     r0, #(ROM_SAVESIZE and 0ffh)
ROMCALL task_genesis            ; Initialize scheduler and create idle thread
```

## int task_getcurrent(void);

*Returns the current task's ID.*

Table 47. **task_getcurrent() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | a | Task ID return value |

Example 28. **task_getcurrent()**

```
ROMCALL task_getcurrent
```

## int task_getpriority(int id);

*Returns the priority of a task.*

This function returns the priority of the task with the given ID. ID 0 means current task. This function returns a success/failure code.

Table 48. **task_getpriority() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | *id* | a | Success/failure code |
| | | b | Task priority return value |

Example 29. **task_getpriority**

```
mov     a, .....
ROMCALL task_getpriority
; b is the priority if a = 0
```

## int task_setpriority(int id, int priority);

*Changes a task priority.*

This function changes the priority of a task. ID 0 means current task. The priority is a value in the range MIN_PRIORITY...MAX_PRIORITY, with an idle task running at MIN_PRIORITY, a regular task running at NORM_PRIORITY. This call returns a success/failure code *(see page 49 for the priority values)*.

Table 49. **task_setpriority() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | *id* | a | Success/failure code |
| b | *priority* | | |

Example 30. **task_setpriority()**

```
clr     a
mov     b, #NORM_PRIORITY-1    ; Only run if all other regular tasks have nothing to do
ROMCALL task_setpriority
```

## int task_fork(int priority, int savesize);

*Creates and executes a new task.*

This function creates a new task and links it into running task list with the given *priority*. The function creates a duplicate of the current task and assigns it a new task id. The new task returns with a zero id, the parent gets the child id as a return value. The function also returns a success/failure code. The function calls mmalloc() to allocate a buffer of *savesize*, which is used to save and restore the task's state. *priority* is a user assigned value in the range MIN_PRIORITY...MAX_PRIORITY, with an idle task running at MIN_PRIORITY, a regular task running at NORM_PRIORITY. The child task is suspended immediately after creation.

Table 50. **task_fork() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | *priority* | a | Success/failure code |
| r1:r0 | *savesize* | r0 | Child task ID or 0 if child |

Example 31. **task_fork()**

```
        ; Create the DHCP process
        mov     r1, #(ROM_SAVESIZE shr 8)
        mov     r0, #(ROM_SAVESIZE and 0ffh)
        mov     a, #NORM_PRIORITY
        ROMCALL task_fork
        jnz     dhcp_init_exit

        mov     a, r0
        jnz     dhcp_init_parent

        ; This is the child process - run the FSM
        ajmp    dhcp_run

dhcp_init_parent:
        ; Record the DHCP task id
        mov     dptr, #DHCP_TASKID
        PUTX
```

## int task_kill(int id);

*Kills a task.*

This function destroys a task and frees its state buffer. ID 0 means current task. The function returns a success/failure code. Note: This function does not interact with the socket code. Call the socket function `cleanup()` to close all associated sockets.

Table 51. **task_kill() register usage**

| Input | Description | Output | Description |
| --- | --- | --- | --- |
| a | *id* | a | Success/failure code |

Example 32. **task_kill()**

```
mov     dptr, #DHCP_TASKID
GETX
ROMCALL task_kill
```

## int task_suspend(int id, int eventmask);

*Suspends a task.*

This function suspends a task until all events in *eventmask* have been generated. ID 0 means current task. The function returns a success/failure code.

Before task suspension, the `switch_out()` hook is called.

Table 52. **task_suspend() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | *id* | a | Success/failure code |
| b | *eventmask* | | |

Example 33. **task_suspend()**

```
clr      a                        ; This task
mov      b, #TASK_DHCPSLEEP
ROMCALL  task_suspend
```

## int task_sleep(int id, int eventmask, int milliseconds);

*Puts a task to sleep.*

Suspends a task until at least *milliseconds* milliseconds have elapsed *or* the event *eventmask* has occurred (use *eventmask* = 0 for regular sleep). ID 0 means current task. Before suspension, the `switch_out()` hook is called.

The function returns a success/failure code.

Table 53. **task_sleep() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | *id* | a | Success/failure code |
| b | *eventmask* | | |
| r3:r0 | *milliseconds* | | |

Example 34. **task_sleep()**

```
clr     a; This task
mov     b, a; Regular sleep for 10 seconds
mov     r3, #(10000 shr 24)
mov     r2, #((10000 shr 16) and 0ffh)
mov     r1, #((10000 shr 8) and 0ffh)
mov     r0, #(10000 and 0ffh)
ROMCALL task_sleep
```

### int task_signal(int id, int eventmask);

*Signals a task.*

This function sends event(s) in *eventmask* to a task. If the task is waiting for no other events, it will wake up and be electable to be run by the task switcher. ID 0 means current task. The function returns a success/failure code.

Table 54. **task_signal() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| a | *id* | a | Success/failure code |
| b | *eventmask* | | |

Example 35. **task_signal()**

```
mov      a, #....                   ; Some task
mov      b, #TASK_USER0             ; User defined event
ROMCALL  task_signal
```

## 6.3.1 Description of Hooks

To allow the user to extended the task scheduler, e.g. to save and restore additional properties of a task, the task scheduler calls “hook” functions. The user must make sure to preserve all registers across the function call.

### task_create

is called when the very first task block is created (genesis). The DS80C400 Silicon Software implementation of this function does nothing.

Table 55. **task_create parameters**

| Register | Description |
|---|---|
| `r1:r0` | *savesize* |
| `dptr0` | newly allocated TCB |

## task_duplicate

is called when the fork operation is called. The DS80C400 Silicon Software implementation of this function does nothing.

Table 56. **task_duplicate parameters**

| Register | Description |
|---|---|
| `r1:r0` | *savesize* |
| `r7` | *priority* |
| `dptr0` | newly allocated TCB |

## task_destroy

is called when a task is destroyed. The DS80C400 Silicon Software implementation of this function does nothing.

Table 57. **task_destroy parameters**

| Register | Description |
|---|---|
| `dptr0` | TCB of process to be removed |

## task_switch_in

is called before a task is switched in. The DS80C400 Silicon Software implementation of this function restores the state in the state buffer (stack and SFRs) and is exported as `rom_task_switch_in`.

Note: This function does not return. However, it returns to the caller of `task_switch_out` with a = 0.

Table 58. **task_switch_in parameters**

| Register | Description |
|---|---|
| `dptr0` | TCB of task to be switched in |

### task_switch_out

is called before a task is suspended (voluntarily because it waits for an event or because its timeslice is over). The DS80C400 Silicon Software implementation of this function saves the task's state (stack and SFRs) to the state buffer and is exported as `rom_task_switch_out`.

Note: This function returns `a` != 0. `task_switch_in` returns to the caller of this function with `a` = 0.

Table 59. **task_switch_out parameters**

| Register | Description |
|---|---|
| `dptr0` | TCB of task to be switched in |

## 7 Utility Functions

The following is a description of all utility functions exported by the DS80C400 Silicon Software.

### int crc16(int crc, unsigned char value);

*Computes the CRC-16 of a byte given an initial CRC value.*

Table 60. **crc16() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| `a` | *value* | `r1:r0` | CRC16 return value |
| `r1:r0` | *crc* | | |

Example 36. **crc()**

```
mov     r1, #0
mov     r0, #0
...
GETX
ROMCALL crc16
```

## void mem_clear(void *target, int length);

*Clears a block of memory.*

Table 61. **mem_clear() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| b:a | *length* | — | |
| dptr0 | *target* | | |

Example 37. **mem_clear()**

```
mov     b, #(MEMLENGTH shr 8)
mov     a, #(MEMLENGTH and 0ffh)
mov     dptr, #MEMTOCLEAR
ROMCALL mem_clear
```

## void mem_copy(void *source, void *target, int length);

*Copies a block of memory.*

Table 62. **mem_copy() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| b:a | *length* | dptr0 | *source+len* |
| dptr0 | *source* | dptr1 | *length+len* |
| dptr1 | *target* | | |

Example 38. **mem_copy()**

```
mov     b, #(MEMLENGTH shr 8)
mov     a, #(MEMLENGTH and 0ffh)
mov     dptr, #SOURCE
inc     dps
mov     dptr, #TARGET
inc     dps
ROMCALL mem_copy
```

## int mem_compare(void *block0, void *block1, int length);

*Compares two blocks of memory.*

This function returns 0 if the two memory blocks are identical, non-zero otherwise.

Table 63. **mem_compare_xdata() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| b:a | *length* | a | Return value |
| dptr0 | *block0* | | |
| dptr1 | *block1* | | |

Example 39. **mem_compare()**

```
mov     b, #(MEMLENGTH shr 8)
mov     a, #(MEMLENGTH and 0ffh)
mov     dptr, #BLOCK0
inc     dps
mov     dptr, #BLOCK1
inc     dps
ROMCALL mem_compare
```

## void add_dptr0(void *dptr0, int value);<br>void add_dptr1(void *dptr1, int value);

*Adds a value to dptr0/1.*

Table 64. **add_dptr0() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| b:a | *value* | dptr0 | Pointer return value |
| dptr0 | Original pointer value | | |

Example 40. **add_dptr0()**

```
mov     b, #01h
clr     a
mov     dptr, #STARTPTR
ROMCALL add_dptr0
```

### void sub_dptr0(void *dptr0, int value);
### void sub_dptr1(void *dptr1, int value);

*Subtracts a value from dptr0/1.*

Table 65. **sub_dptr0() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| b:a | *value* | dptr0 | Pointer return value |
| dptr0 | Original pointer value | | |

Example 41. **sub_dptr0()**

```
mov     b, #01h
clr     a
mov     dptr, #ENDPTR
ROMCALL sub_dptr0
```

### unsigned char getpseudorandom(void);

*Gets a pseudorandom byte from a CRC function.*

Table 66. **getpseudorandom() register usage**

| Input | Description | Output | Description |
|---|---|---|---|
| — | | a | Return value |

Example 42. **getpseudorandom()**

```
ROMCALL getpseudorandom
```

## 8 Using DS80C400 Silicon Software Functions

Since the memory location of the DS80C400 Silicon Software functions can change between DS80C400 Silicon Software revisions, the following technique must be used to acquire a function address.

1. Read the 24-bit address of the Export Table from memory address `FF0002`. This address is stored most significant byte first, e.g. `FF 28 44`.
2. Check the index number against the *Num_Fn* parameter (see Appendix A).

3. Get the address of the desired function/structure: *addr = ExportTable[index * 3]*. (this value is also stored most significant byte first, e.g. `FF 01 7B`).

Appendix A lists the index numbers of all currently exported functions. New functions will be added to the bottom of this list.

## 9 DS80C400 Silicon Software Version

A zero-terminated ASCII string at the location `FF0005` describes the DS80C400 Silicon Software revision.

Example 43. **DS80C400 Silicon Software Version**

```
TINI400-1.0.0
```

# Appendix A DS80C400 Silicon Software Export Table

Table 67. **DS80C400 Silicon Software Export Table**

| Index | Name | Description / Group |
|---|---|---|
| 0 | (Num_Fn, 0, 0) | 100 — number of functions following |
| 1 | crc16 | Utility functions—see Section 7 |
| 2 | mem_clear | |
| 3 | mem_copy | |
| 4 | mem_compare | |
| 5 | add_dptr0 | |
| 6 | add_dptr1 | |
| 7 | sub_dptr0 | |
| 8 | sub_dptr1 | |
| 9 | getpseudorandom | |
| 10 | rom_kernelmalloc | DS80C400 Silicon Software memory manager |
| 11 | rom_kernelfree | |
| 12 | rom_malloc | |
| 13 | rom_malloc_dirty | |
| 14 | rom_free | |
| 15 | rom_deref | |
| 16 | rom_getfreeram | |

Table 67. **DS80C400 Silicon Software Export Table**

| Index | Name | Description / Group |
|---|---|---|
| 17 | socket | Socket functions—see Section 5 |
| 18 | closesocket | |
| 19 | sendto | |
| 20 | recvfrom | |
| 21 | connect | |
| 22 | bind | |
| 23 | listen | |
| 24 | accept | |
| 25 | recv | |
| 26 | send | |
| 27 | getsockopt | |
| 28 | setsockopt | |
| 29 | getsockname | |
| 30 | getpeername | |
| 31 | cleanup | |
| 32 | avail | |
| 33 | join | |
| 34 | leave | |
| 35 | ping | |
| 36 | getnetworkparams | |
| 37 | setnetworkparams | |
| 38 | getipv6params | |
| 39 | getethernetstatus | |
| 30 | gettftpserver | |
| 41 | settftpserver | |
| 42 | eth_processinterrupt | Default Ethernet interrupt handler |
| 43 | arp_generaterequest | Generates ARP request |
| 44 | MAC_ID | Pointer to MAC ID |

Table 67. **DS80C400 Silicon Software Export Table**

| Index | Name | Description / Group |
|---|---|---|
| 45 | dhcp_init | DHCP functions—see Section 5.4.7 |
| 46 | dhcp_setup | |
| 47 | dhcp_startup | |
| 48 | dhcp_run | |
| 49 | dhcp_status | |
| 50 | dhcp_stop | |
| 51 | rom_dhcp_notify | |
| 52 | tftp_init | TFTP functions—see Section 5.4.7 |
| 53 | tftp_first | |
| 54 | tftp_next | |
| 55 | TFTP_MSG | |
| 56 | task_genesis | Task scheduler functions—see Section 6 |
| 57 | task_getcurrent | |
| 58 | task_getpriority | |
| 59 | task_setpriority | |
| 60 | task_fork | |
| 61 | task_kill | |
| 62 | task_suspend | |
| 63 | task_sleep | |
| 64 | task_signal | |
| 65 | rom_task_switch_in | |
| 66 | rom_task_switch_out | |
| 67 | EnterCritSection | Enter/Leave critical section. |
| 68 | LeaveCritSection | |

Table 67. **DS80C400 Silicon Software Export Table**

| Index | Name | Description / Group |
|---|---|---|
| 69 | rom_init | Initialization functions—see Appendix B |
| 70 | rom_copyivt | |
| 71 | rom_redirect_init | |
| 72 | mm_init | |
| 73 | km_init | |
| 74 | ow_init | |
| 75 | network_init | |
| 76 | eth_init | |
| 77 | init_sockets | |
| 78 | tick_init | |
| 79 | WOS_Tick | Default timer interrupt handler—see Appendix C |
| 80 | BLOB | Start address of memory area ignored by NetBoot—see Section 3 |
| 81 | WOS_IOPoll | Asynchronous TCP/IP maintenance functions |
| 82 | IP_ProcessReceiveQueues | |
| 83 | IP_ProcessOutput | |
| 84 | TCP_RetryTop | |
| 85 | ETH_ProcessOutput | |
| 86 | IGMP_GroupMaintainence | |
| 87 | IP6_ProcessReceiveQueues | |
| 88 | IP6_ProcessOutput | |
| 89 | PARAMBUFFER | Pointer to parameter buffer—see Section 5.4.5 |
| 90 | RAM_TOP | Address of pointer to end of RAM used by NetBoot |
| 91 | BOOT_MEMBEGIN | |
| 92 | BOOT_MEMEND | |
| 93 | OWM_First | 1-Wire master functions |
| 94 | OWM_Next | |
| 95 | OWM_Reset | |
| 96 | OWM_Byte | |
| 97 | OWM_Search | |
| 98 | OW_ROMID | |

Table 67. **DS80C400 Silicon Software Export Table**

| Index | Name | Description / Group |
|---|---|---|
| 99 | AutoBaud | Serial port 0 baud rate detection |
| 100 | tftp_close | TFTP function—see Section 5.4.7 |

# Appendix B Initializing the DS80C400 Silicon Software System

Before using any DS80C400 Silicon Software function, the system must have a valid interrupt vector table and function redirect table, and all memory must be cleared (directs, RAM). To use the DS80C400 Silicon Software interrupt vector table, call `ROM_CopyIVT`. To use the DS80C400 Silicon Software function redirect table, call `ROM_Redirect_Init`.

To use the network and task functions, the system must be initialized using the *_init functions in the following order:

1. `MM_Init` Initialize heap
2. `KM_Init` Initialize memory
3. `OW_Init` Initialize 1-Wire
4. `Network_Init` Network layer
5. `ETH_Init` Network driver
6. `SOCK_Init` Socket layer
7. `Tick_Init` Timer interrupt
8. `task_genesis` Scheduler and idle thread
9. `setb ea` Enable interrupts

The function call `ROM_Init` takes care of all required initialization (including interrupt vector table and function redirect table). `ROM_Init` also prints a copyright message.

# Appendix C Using a 1-Wire Chip for Static IP Address Configuration

The DS80C400 Silicon Software Environment supports the following 1-Wire parts as IP address configuration source: DS1427, DS1971, DS1973, DS1992, DS1993, DS1994, DS1995, DS1996, DS2404, DS2430A, DS2433 and DS2504.

The first part that contains the following signature at offset 0 is queried for IPv4 address, IPv4 gateway address, IPv4 prefix length (will be converted to the subnet mask) and TFTP server IP (this IP is also supported for IPv6[1]):

Table 68. **1-Wire Address Configuration Data**

| Offset | | Description |
|---|---|---|
| 0 | 29 | Length byte |
| 1 | 'TINI' | Signature |
| 5 | 32-bit value | IPv4 |
| 9 | 32-bit value | IPv4 gateway |
| 13 | 1 byte | IPv4 prefix length (number of 1-bits in subnet mask) |
| 14 | 128-bit value | TFTP server IP |
| 30 | 16-bit value | 1-complement of CRC-16 (LSB first) |

The data is organized in standard 1-Wire format (length byte—data—CRC-16).

The following Java code is a simplified example which demonstrates how to write this information to a DS2433. Production code should verify the data after the READ_SCRATCHPAD command to ensure that the data was received correctly before committing it to flash.

Example 44. **Programming the DS2433 to Hold the IP Address**

```
import com.dalsemi.onewire.adapter.TINIInternalAdapter;
import com.dalsemi.onewire.OneWireException;
import com.dalsemi.onewire.utils.CRC16;

class WriteIP {
    static final int TARGET_FAMILY_ID         = 0x23;
    static final int READ_MEMORY_COMMAND      = 0xf0;
```

1. If the first 12 bytes are 0, it is an IPv4 address, else an IPv6 address.

```
static final int WRITE_SCRATCHPAD_COMMAND = 0x0f;
static final int COPY_SCRATCHPAD_COMMAND  = 0x55;
static final int READ_SCRATCHPAD_COMMAND  = 0xaa;

public static void main(String[] args) {
        TINIInternalAdapter adapter = new TINIInternalAdapter();
        boolean foundIt = false;

        try {
                long deviceAddress;

                adapter.beginExclusive(true);
                if (adapter.findFirstDevice()) {
                        // Test LSB (family id) against target
                        deviceAddress = adapter.getAddressAsLong();
                        if ((deviceAddress & 0xff) == TARGET_FAMILY_ID) {
                                foundIt = true;
                        }
                        while (!foundIt && adapter.findNextDevice()) {
                                deviceAddress = adapter.getAddressAsLong();
                                if ((deviceAddress & 0xff) == TARGET_FAMILY_ID) {
                                        foundIt = true;
                                }
                        }
                        if (foundIt) {
                                byte[] command = new byte[4];
                                byte[] config = new byte[32];
                                config[0] = 29;// length
                                config[1] = 'T';// signature
                                config[2] = 'I';
                                config[3] = 'N';
                                config[4] = 'I';
                                config[5] = (byte) 192;//IPv4
                                config[6] = (byte) 168;
                                config[7] = (byte) 0;
                                config[8] = (byte) 1;
                                config[9] = (byte) 192;// IPv4 gateway
                                config[10] = (byte) 168;
                                config[11] = (byte) 0;
                                config[12] = (byte) 2;
                                config[13] = 24;// IPv4 prefix length
                                config[14] = 0;// TFTP server IP (16 bytes)
                                config[15] = 0;
                                config[16] = 0;
                                config[17] = 0;
                                config[18] = 0;
                                config[19] = 0;
                                config[20] = 0;
                                config[21] = 0;
                                config[22] = 0;
                                config[23] = 0;
                                config[24] = 0;
                                config[25] = 0;
                                config[26] = (byte) 192;
                                config[27] = (byte) 168;
                                config[28] = (byte) 0;
                                config[29] = (byte) 3;

                                crc = ~CRC16.compute(config, 0, 30);
                                config[30] = (byte) (crc & 0xff);
                                config[31] = (byte) ((crc >> 8) & 0xff);

                                command[0] = (byte) WRITE_SCRATCHPAD_COMMAND;
```

```
                                        command[1] = 0;
                                        command[2] = 0;
                                        adapter.select(deviceAddress);
                                        adapter.dataBlock(command, 0, 3);
                                        adapter.dataBlock(config, 0, 32);

                                        command[0] = (byte) READ_SCRATCHPAD_COMMAND;
                                        adapter.select(deviceAddress);
                                        adapter.dataBlock(command, 0, 1);
                                        byte[] scratch = adapter.getBlock(3+32);

                                        // VERIFY DATA HERE

                                        command[0] = (byte) COPY_SCRATCHPAD_COMMAND;
                                        command[1] = 0;
                                        command[2] = 0;
                                        command[3] = scratch[2];
                                        adapter.select(deviceAddress);
                                        adapter.dataBlock(command, 0, 4);

                                } else {
                                        System.out.println("Device not found");
                                }
                        }
                } catch (OneWireException owe) {
                        System.out.println(owe.getMessage());
                } finally {
                        adapter.endExclusive();
                }
        }
    }
```

## Appendix D  Site-specific DHCP Option 150 (TFTP Server IP)

Since some DHCP servers prevent configuration of the ‘next server IP’ field, the DS80C400 Silicon Software Environment uses the site-specific (user defined) option 150 if present. If the option is present, it always overrides the ‘next server IP’ value.

This option is defined as follows (see RFC 2132 for the description of standard DHCP options):

Table 69. **Site-specific Option 150**

| Code | Len | Address | | | |
|---|---|---|---|---|---|
| 150 | 4 | a1 | a2 | a3 | a4 |

To configure option 150 in ISC *dhcpd*, use

```
option option-150 aa:bb:cc:dd;
```

where `aa:bb:cc:dd` is the hexadecimal representation of the TFTP server IP address, e.g. `192.168.0.3` would be configured as

```
option option-150 c0:a8:00:03;
```

To configure option 150 on Windows 2000, see *http://www.cisco.com/warp/public/788/AVVID/win2000_dhcp.html*

## Appendix E Using the Keil C Compiler

Functions in the DS80C400 Silicon Software have been exposed to allow developers using the Keil C Compiler to access them. Make sure when working with the C compiler that you include the file **startup400.a51**, instead of the file **startup.a51** that comes with the Keil C compiler. To use functions in the DS80C400 Silicon Software, include the header file that contains definitions for the functions you want to use, and add the assembler file that contains the wrappers to those ROM functions to your project.

| Header File | Assembler File | Description of functions |
|---|---|---|
| rom400_dhcp.h | rom400_dhcp.a51 | DHCP Client |
| rom400_init.h | rom400_init.a51 | ROM Initialization |
| rom400_mem.h | rom400_mem.a51 | Memory management |
| rom400_ow.h | rom400_ow.a51 | 1-Wire communications |
| rom400_sched.h | rom400_sched.a51 | Task and Thread management |
| rom400_sock.h | rom400_sock.a51 | Socket communications |
| rom400_tftp.h | rom400_tftp.a51 | TFTP communication |
| rom400_util.h | rom400_util.a51 | CRC16, GetRandom, Mem_Cmp |

More detailed information on the available functions can be found in the header files. Before using any ROM functions, make sure to call the `rom_init` function (declared in header file **rom400_init.h**) to initialize the DS80C400 Silicon Software.

# Appendix F Error Codes

# Appendix G Bibliography

Dallas Semiconductor, *The DS80C400 Datasheet.*

# Appendix H Document Revision History

Table 70. **Revision History**

| **Release** | **Comments** |
|---|---|
| 1 | Initial Release |
| 2 | DS80C400 Rev. B1 Changes, Keil C documentation |

The User's Guide was updated to reflect the following changes in the B1 ROM revision:

## General

- The ROM software runs without clock doubler and sets MOVX stretch cycles to maximum
- Merged current TINI software fixes and network stack changes and improved the DS80C400 Ethernet driver
- Use DS80C400 hardware acceleration for 1-Wire and IP checksums
- The software will enter sleep mode if no network and no user code can be found

## Serial Loader

- Revised copyright notice and moved serial number display to company standard representation
- Changed baud rate detection to work with all clock speeds at all baud rates (within the serial port hardware constraints)
- Serial loader supports loading into SRAM

- The dump command displays an Intel 386 record for the high address byte
- Eliminated the 'S' command

## Network loader

- Serial output will only be present when started from Serial Loader
- 1-Wire timing and network delays are derived from the DS2502 1-Wire bit widths
- The DHCP delays were increased
- The TFTP code randomizes its port numbers

## ROM software support

- Exported the 1-Wire master subsystem and the following:
- IOPoll User Extensions, Error Notification, AutoBaud, memory boundaries, and a binary revision number