TargetTCP Porting Guide

TargetTCP is developed and maintained using TargetOS, but is easily ported to other operating systems or to polling environments. Except for an optional assembly language checksum routine (an alternative C routine is provided), the source code is 100% ANSI C. Minimal RTOS dependences exist. This guide identifies the dependencies that do exist and provides guidelines for porting. Examples are used from existing TargetTCP ports to Nucleus and pSOS+.

The following is a list of the RTOS dependencies:

- Task creation and deletion
- Semaphores to protect critical sections and provide reentrancy
- Support for calling a timing function ten times per second
- Methods to mask and unmask CPU interrupts
- A method to get the ID of the currently running task
- A method for a task or ISR to wake a blocked task, given its task ID
- A method for a task to block until wakened by another task or ISR
- A means for equating time values with RTOS ticks

The documentation below covers each of the dependencies listed above. Various RTOS and 'glue' functions that must be considered during the port are discussed. In many cases, the best porting approach is to write wrapper functions that implement the TargetOS service calls using the RTOS being ported to, so it “looks” like TargetOS to the TCP/IP code.

A few routines, however, are so frequently called and whose translation is so direct that it is better to port them via #defines rather than function wrappers. The documentation below identifies these functions. 'tcp_ipp.h' is a convenient place to put the needed macro definitions.

Example function wrappers and macro definitions for the necessary service calls are provided in the directory 'extos'. These serve as a starting point for new RTOS ports.

Tasks

TargetTCP uses one task for its main protocol daemon and one task for every session created by the optional FTP and Telnet components. The priority of the protocol daemon task should be higher than any task that calls the TargetTCP API routines. This section covers the calls used by TargetTCP to create and delete tasks, and how they can be ported to other operating systems.

`taskCreate()` - Used to create the TargetTCP daemon task during TargetTCP initialization and at the start of a new FTP and Telnet server session.

An example implementation for Nucleus is:

```
#define TASK_PRIORITY   2
#define STACK_LWORDS    1024            /* targettcpd stack size */
```


6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

```
static void (*TaskCallBack)(ui32 unused);
static ULONG TaskStack[STACK_LWORDS];
static NU_TASK Task;

/************************************************************************/
/*  targettcpd: Implement TargetTCP protocol daemon                     */
/*                                                                      */
/************************************************************************/
static void targettcpd(UNSIGNED argc, void *argv)
{
  TaskCallBack(0);
}

/************************************************************************/
/*  taskCreate: Nucleus implementation                                  */
/*                                                                      */
/************************************************************************/
TASK taskCreate(const char name[8], ui8 stack_size, ui8 priority,
                void (*funcp)(ui32 msg), ui32 msg, ui32 flags)
{
  STATUS status;

  /*----------------------------------------------------------------*/
  /* Save callback pointer to daemon code.                          */
  /*----------------------------------------------------------------*/
  TaskCallBack = funcp;

  /*----------------------------------------------------------------*/
  /* Create the TargetTCP daemon task.                              */
  /*----------------------------------------------------------------*/
  status = NU_Create_Task(&Task, "tcpd", targettcpd, 0, NULL,
                          (void *)TaskStack, 4 * STACK_LWORDS,
                          TASK_PRIORITY, 0, NU_PREEMPT, NU_START);
  NetAssert(status == NU_SUCCESS);

  /*----------------------------------------------------------------*/
  /* Return task identifier.                                        */
  /*----------------------------------------------------------------*/
  return &Task;
}
```

An example implementation for pSOS+ is:

```
/************************************************************************/
/*  task_start: task start function with auto-delete upon return       */
/*                                                                      */
/************************************************************************/
static void task_start(ui32 arg0, ui32 arg1, ui32 arg2, ui32 arg3)
{
  void (*funcp)(ui32 msg);

  funcp = (void (*)(ui32 msg))arg1;
  funcp(arg0);
  t_delete(0); /* 'automatic' deletion like TargetOS */
```

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

```
}

/*************************************************************************/
/*  taskCreate: pSOS+ implementation                                     */
/*                                                                       */
/*************************************************************************/
TASK taskCreate(const char name[8], ui8 stack_size,
            ui8 priority, void (*funcp)(ui32 msg), ui32 msg, ui32 flags)
{
  ui32 t_create(char name[4], ui32 prio, ui32 sstack, ui32 ustack,
                ui32 flags, ui32 *tid);
  ui32 t_start(ui32 tid, ui32 mode, void (*start_addr)(), ui32 targs[]);
  ui32 tid, rc, arg[4];

  /*---------------------------------------------------------------------*/
  /* Call t_create() to create task.                                     */
  /*---------------------------------------------------------------------*/
  if (rc = t_create((char *)name, priority, DAEMON_STACK, 0, 0, &tid))
  {
    errno = rc;
    return 0;
  }

  /*---------------------------------------------------------------------*/
  /* Initialize its parameters and call t_start() to start task.         */
  /*---------------------------------------------------------------------*/
  arg[0] = msg;
  arg[1] = (ui32)funcp;
  if (rc = t_start(tid, T_SUPV, task_start, arg))
  {
    errno = rc;
    return 0;
  }

  /*---------------------------------------------------------------------*/
  /* Return task identifier.                                             */
  /*---------------------------------------------------------------------*/
  return tid;
}
```

`taskDelete()` - Called to delete the daemon task if TargetTCP initialization fails after the daemon task has been created. Also, FTP and Telnet server tasks delete themselves by returning from their implementation function after the session is finished. This is supported by TargetOS. Ports to other RTOS's can add a taskDelete() call at the end of the FTP server function ftpd() or use the auto-delete method in the pSOS+ task_start() above.

An example implementation for Nucleus is:

```
/*************************************************************************/
/*  taskDelete: Nucleus implementation                                   */
/*                                                                       */
/*************************************************************************/
int taskDelete(TASK *tidp)
{
```

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

```
  /*-------------------------------------------------------------------*/
  /* Call NU_Delete_Task() to delete task.                             */
  /*-------------------------------------------------------------------*/
  return NU_Delete_Task(*tidp);
}
```

An example implementation for pSOS+ is:

```
/*********************************************************************/
/*  taskDelete: pSOS+ implementation                                 */
/*                                                                   */
/*********************************************************************/
int taskDelete(TASK *tidp)
{
  ui32 t_delete(ui32 tid);

  /*-------------------------------------------------------------------*/
  /* Call t_delete() to delete task.                                   */
  /*-------------------------------------------------------------------*/
  return t_delete(*tidp);
}
```

Semaphores

Semaphores are used by TargetTCP to protect critical sections. Two semaphores are used per socket (for full-duplex access) and one each is used to protect TargetTCP global variables, protect access to the DNS client, and protect access to the TFTP client. Only binary semaphores are required, but counting semaphores can be used. These can be implemented using the counting semaphores that virtually all operating systems provided.

`semCreate()` - Called at startup during TargetTCP initialization.

An example implementation for Nucleus is:

```
#define NUM_SEM         (3 + 2 * MAX_NUM_SOCKS)
static NU_SEMAPHORE SemCtrlBlk[NUM_SEM];
static ui8 SemBlkUsed[NUM_SEM];

/*********************************************************************/
/*   semCreate: Nucleus counting semaphore creation                  */
/*                                                                   */
/*      Inputs: name = pointer to name string (up to 8 chars)        */
/*              count = initial token count                          */
/*              mode = unused                                        */
/*                                                                   */
/*        Note: Only called at startup or with internal access       */
/*                                                                   */
/*********************************************************************/
SEM semCreate(const char *name, int count, int unused)
{
  int i;
  STATUS status;
```

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

```
  NU_SEMAPHORE *semaphore;

  /*-------------------------------------------------------------------*/
  /* Allocate semaphore control block.                                 */
  /*-------------------------------------------------------------------*/
  for (i = 0;; ++i)
  {
    NetAssert(i < NUM_SEM);
    if (SemBlkUsed[i] == FALSE)
    {
      SemBlkUsed[i] = TRUE;
      semaphore = &SemCtrlBlk[i];
      break;
    }
  }

  /*-------------------------------------------------------------------*/
  /* Create FIFO-mode counting semaphore.                              */
  /*-------------------------------------------------------------------*/
  status = NU_Create_Semaphore(semaphore, (char *)name, count, NU_FIFO);
  NetAssert(status == NU_SUCCESS);

  /*-------------------------------------------------------------------*/
  /* Return pointer to semaphore control block.                        */
  /*-------------------------------------------------------------------*/
  return semaphore;
}
```

An example implementation for pSOS+ is:

```
/*********************************************************************/
/*   semCreate: pSOS+ implementation                                 */
/*                                                                   */
/*********************************************************************/
SEM semCreate(const char name[8], int init_count, int mode)
{
  ui32 sm_create(char name[4], ui32 count, ui32 flags, ui32 *smid);
  ui32 sem, rc = sm_create((char *)name, init_count, SM_FIFO, &sem);

  /*-------------------------------------------------------------------*/
  /* Return -1 and set errno if error, else return identifier.         */
  /*-------------------------------------------------------------------*/
  if (rc)
  {
    errno = rc;
    return 0;
  }
  else
  {
    return sem;
  }
}
```

`semDelete()` - Called if TargetTCP initialization fails after one or more semaphores have already been created.

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

An example implementation for Nucleus is:

```
/************************************************************************/
/*   semDelete: Nucleus semaphore deletion                              */
/*                                                                      */
/************************************************************************/
void semDelete(SEM *semp)
{
  int i;
  SEM semaphore = *semp;
  STATUS rc;

  /*----------------------------------------------------------------------*/
  /* Delete the semaphore.                                                */
  /*----------------------------------------------------------------------*/
  rc = NU_Delete_Semaphore(semaphore);
  NetAssert(rc == NU_SUCCESS);

  /*----------------------------------------------------------------------*/
  /* Find and free the semaphore control block.                           */
  /*----------------------------------------------------------------------*/
  for (i = 0;; ++i)
  {
    NetAssert(i < NUM_SEM);
    if (semaphore == &SemCtrlBlk[i])
    {
      SemBlkUsed[i] = FALSE;
      *semp = 0;
      return;
    }
  }
}
```

An example implementation for pSOS+ is:

```
/************************************************************************/
/*   semDelete: pSOS+ implementation                                    */
/*                                                                      */
/************************************************************************/
void semDelete(SEM *semp)
{
  SEM sem = *semp;
  ui32 sm_delete(ui32 smid);
  ui32 rc = sm_delete(sem);

  *semp = 0;
  NetAssert((rc == 0) || (rc == 0x44));
}
```

`semPend()` - Used to wait for access to a resource, such as a socket, the DNS client, or TargetTCP internal variables. The wait option is always WAIT_FOREVER, but since no task ever blocks while holding onto the TargetTCP critical section semaphore, the semPend() wait is usually short.

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

An example implementation for Nucleus is:

```
STATUS NU_Obtain_Semaphore(NU_SEMAPHORE *sem, UNSIGNED suspend);
#define semPend(sem, waitopt)   \
  NU_Obtain_Semaphore(sem, 0xFFFFFFFD) != NU_SUCCESS
```

An example implementation for pSOS+ is:

```
ui32 sm_p(ui32 smid, ui32 flags, ui32 timeout);
#define SemPend(sem, wait_opt)   sm_p((ui32)(sem), SM_WAIT, wait_opt)
```

`semPost()` - Used to release access to a resource, such as a socket, the DNS client, or TargetTCP internal variables.

An example implementation for Nucleus is:

```
STATUS NU_Release_Semaphore(NU_SEMAPHORE *semaphore);
#define semPost(sem)    NU_Release_Semaphore(sem)
```

An example implementation for pSOS+ is:

```
ui32 sm_v(ui32 sem);
#define semPost(sem)    sm_v((ui32)(sem))
```

These declarations and definitions can be added to "tcp_ipp.h".

Timers

TargetTCP has one timer function, netTick(), that drives all timer-related processing. The criteria is that netTick() must be called ten times a second. If the RTOS being ported to has good timer support, this can be done using an RTOS timer. Alternately, the call can be made in the interrupt service routine of a hardware timer or a task can be created that repeatedly sleeps for 100 milliseconds after making the call.

TargetTCP uses tmrCreate() to create a TargetOS timer and tmrCallAfter() to use it to schedule the periodic netTick() calls. An implementation for pSOS+ is:

```
/*********************************************************************/
/*   tcp_timer: TargetTCP timer task                                 */
/*                                                                   */
/*********************************************************************/
static void tcp_timer(ui32 ticks)
{
  for (;;)
  {
    netTick();
    tm_wkafter(ticks);
  }
}
```

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

```
/************************************************************************/
/* Global Function Definitions                                          */
/************************************************************************/

/************************************************************************/
/* tmrCallAfter: Call user function after specified tick count          */
/*                                                                      */
/*      Inputs: tick = number of ticks before call                      */
/*              func = points to function to call                       */
/*              arg = argument to call function with                    */
/*              timer = handle for timer to use for scheduling call     */
/*                                                                      */
/************************************************************************/
int tmrCallAfter(int ticks, int (*func)(ui32 arg), ui32 arg, TMR timer)
{
  ui32 targs[4], rc;

  /*--------------------------------------------------------------------*/
  /* Start the timer task with the passed in parameters.                */
  /*--------------------------------------------------------------------*/
  targs[0] = ticks;
  targs[1] = (ui32)func;
  targs[2] = arg;
  rc = t_start(timer, T_SUPV | T_NOPREEMPT | T_NOISR, tcp_timer, targs);
  NetAssert(rc == 0);
  return 0;
}

/************************************************************************/
/*   tmrCreate: Create a timer                                          */
/*                                                                      */
/************************************************************************/
TMR tmrCreate(const char name[8])
{
  ui32 rc, tid;

  /*--------------------------------------------------------------------*/
  /* Create the timer task.                                             */
  /*--------------------------------------------------------------------*/
  rc = t_create((char *)name, TMR_PRI, TIMER_STACK, 0, 0, &tid);
  NetAssert(rc == 0);

  return tid;
}
```

Interrupts

Interrupts are masked for a few lines in the buffer allocate and free routines, and briefly in the routines that post messages to and receive messages from the protocol task queue. Consequently, a port must provide routines to mask and unmask interrupts. The TargetOS routines are “isrDisable()” and “isrRestore()”.

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

These functions are used in pairs to temporarily mask processor interrupts. IsrDisable() should mask all processor interrupts. Because these functions may be called from interrupt service routines, isrRestore() can not unconditionally unmask all processor interrupts. Instead, it must restore the interrupt mask present at the isrDisable() call.

Example implementations for Nucleus are:

```
INT NU_Control_Interrupts(INT new_level);
#define isrDisable()        NU_Control_Interrupts(NU_DISABLE_INTERRUPTS)
#define isrRestore(imask)   NU_Control_Interrupts(imask)
```

Example implementations for pSOS+/TriMedia are:

```
UInt IntClearIEN(void);
Void IntRestureIEN(UInt ien);
#define isrDisable()        intClearIEN()
#define isrRestore(imask)   intRestoreIen(imask)
```

These declarations and definitions can be added to "tcp_ipp.h".

Task ID

For task synchronization, TargetTCP needs to be able to read the ID of the currently running task. This is used for waking the task later, after it has blocked on an event, when either the event it is waiting for has occurred or a timeout has expired.

TargetOS stores the ID of the currently running task in the global variable RunningTask. If the RTOS being ported to stores the ID of the running task in a global variable, RunningTask can be renamed to that variable using a #define in "tcp_ipp.h". Otherwise, a #define can be used to rename RunningTask to the name of the appropriate function.

An example implementation for Nucleus is:

```
#define RunningTask   (TASK)NU_Current_Task_Pointer()
```

An example implementation for pSOS+ includes the following in "tcp_ipp.h":

```
#define RunningTask   t_ident_call()
TASK t_ident_call(void);
```

And in "psosintf.c":

```
TASK t_ident_call(void)
{
  ui32 tid;

  t_ident(0, 0, &tid);
  return (TASK)tid;
}
```

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

Task Event Wake

TargetTCP calls taskWake() to wake a task when the event it is waiting for (data arrival, room in a socket send buffer, etc.) has occurred. The function is called with the ID of the task to be wakened as its single parameter. taskWake() can be implemented using any event mechanism supported by the RTOS being ported to.

An example implementation for Nucleus is:

```
#define taskWake(tid)           NU_Resume_Driver((void *)(tid))
```

An example implementation for pSOS+ is:

```
#define TARGETTCP_EVENT         2
#define taskWake(tid)           ev_send((ui32)(tid), TARGETTCP_EVENT)
```

These definitions can be added to the TargetTCP private include file "tcp_ipp.h".

Task Event Wait

TargetTCP uses netEventWait() to have a task block until an event or timeout has occurred. Its single parameter is the maximum number of kernel ticks to wait. This function is only called by tasks that already hold exclusive access to TargetTCP internals, having acquired the internals access semaphore.

In addition to waiting for the event posted by taskWake(), netEventWait() must release the internals access semaphore before blocking and re-acquire it after being woken, without allowing the possibility of event being missed because it was posted after exclusive access was released and before the task has block.

How this should be done for the RTOS being ported to depends on things like whether events are 'sticky' (persistent) and whether the interrupt masking call masks interrupts for all tasks or just the running task (which means interrupts are re-enabled once the running task blocks).

The implementation for TargetOS is:

```
/*************************************************************************/
/* netEventWait: Indivisibly release exclusive access to TargetTCP     */
/*               internals and wait for task wake                        */
/*                                                                       */
/*       Input: wait_opt = maximum number of kernel ticks to wait        */
/*                                                                       */
/*************************************************************************/
void netEventWait(ui32 wait_opt)
{
  /*---------------------------------------------------------------------*/
  /* Ensure task is not preempted before taskSleep().                    */
  /*---------------------------------------------------------------------*/
  taskLock();
```

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

```
  /*-------------------------------------------------------------------*/
  /* Same as netExclPost().                                            */
  /*-------------------------------------------------------------------*/
  Net.Entries = 0;
  semPost(Net.IntSem);

  /*-------------------------------------------------------------------*/
  /* Wait for event or timeout.                                        */
  /*-------------------------------------------------------------------*/
  taskSleep(wait_opt);

  /*-------------------------------------------------------------------*/
  /* Same as netExclPend().                                            */
  /*-------------------------------------------------------------------*/
  semPend(Net.IntSem, WAIT_FOREVER);
  Net.Owner = RunningTask;
  Net.Entries = 1;
}
```

An example implementation for pSOS+ is:

```
/*********************************************************************/
/* netEventWait: Indivisibly release exclusive access to TargetTCP   */
/*               internals and wait for task wake                    */
/*                                                                   */
/*       Input: wait_opt = maximum number of kernel ticks to wait    */
/*                                                                   */
/*********************************************************************/
void netEventWait(ui32 wait_opt)
{
  ui32 events_r;

  /*-------------------------------------------------------------------*/
  /* Prepare for ev_receive() by clearing any pre-existing event.      */
  /*-------------------------------------------------------------------*/
  ev_receive(TARGETTCP_EVENT, EV_NOWAIT | EV_ANY, 0, &events_r);

  /*-------------------------------------------------------------------*/
  /* Same as netExclPost().                                            */
  /*-------------------------------------------------------------------*/
  Net.Entries = 0;
  sm_v(Net.IntSem);

  /*-------------------------------------------------------------------*/
  /* Wait for event or timeout.                                        */
  /*-------------------------------------------------------------------*/
  ev_receive(TARGETTCP_EVENT, EV_WAIT | EV_ANY, wait_opt, &events_r);

  /*-------------------------------------------------------------------*/
  /* Same as netExclPend().                                            */
  /*-------------------------------------------------------------------*/
  sm_p(Net.IntSem, SM_WAIT, WAIT_FOREVER);
  t_ident(0, 0, &Net.Owner);
  Net.Entries = 1;
}
```

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

RTOS Tick Frequency

TargetTCP uses OsTicksPerSec for equating time values with RTOS ticks. OsTicksPerSec is a global variable of type "unsigned long" that holds the number of 'ticks' per second used by the underlying RTOS for task waits. This is used to calculate the correct tick wait option when tasks block in TargetTCP waiting for an event to occur.

An example implementation for Nucleus is:

```
ui32 OsTicksPerSec = 100;
```

An example implementation for pSOS+ is:

```
#include <psoscfg.h>
extern pSOS_CT PsosCfg;

  /*-------------------------------------------------------------------*/
  /* Initialize TargetTCP's RTOS ticks per second variable.            */
  /*-------------------------------------------------------------------*/
  OsTicksPerSec = PsosCfg.kc_ticks2sec;
```

Task Sleep

TargetTCP uses taskSleep() to have a task block until a timeout has occurred. The number of kernel ticks to sleep is its single parameter.

An example implementation for pSOS+ is:

```
#define taskSleep(ticks)        tm_wkafter(ticks)
ui32 tm_wkafter(ui32 ticks);
```

This declaration and definition can be added to "tcp_ipp.h".

SetErrno() and GetErrno()

In some systems, errno is a global variable. In others, it is a macro that references a task control block entry. To isolate errno dependencies, all TargetTCP accesses to errno are done through the functions **GetErrno()** and **SetErrno(),** which need to be implemented for your target environment.

The implementations below are adequate, but they must be built with your environment's "errno.h", so that they use the errno mechanism provided by your environment. These routines should not be included in the library you are making to contain TargetTCP. They should be in the application or some other support code.

```
/*********************************************************************/
/*    SetErrno: Set errno value                                      */
/*                                                                   */
/*       Input: err_val = new value for errno                        */
```

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

```
/*                                                                      */
/************************************************************************/
void SetErrno(int err_val)
{
  errno = err_val;
}

/************************************************************************/
/*    GetErrno: Return errno value                                      */
/*                                                                      */
/************************************************************************/
int GetErrno(void)
{
  return errno;
}
```

Configuration Variables

There are several global variables whose definitions are used to configure TargetTCP. Normally each application has its own copy of these definitions. These configuration variables are described below. The porting example file 'config.c' in the directory 'extos' contains example definitions.

`PriDnsServer and SecDnsServer` - Used to define the IP addresses of the primary and secondary DNS servers if the network interface uses manual IP assignment. If DHCP is used, the primary and secondary DNS server addresses are provided by the DHCP server and the values assigned to PriDnsServer and SecDnsServer are ignored.

```
ui32 PriDnsServer = htonl(0);           /* primary DNS server */
ui32 SecDnsServer = htonl(0);           /* secondary DNS server */
```

`DefGateway` - Used to define the IP address of the default gateway if the network interface uses manual IP assignment. If DHCP is used, the default gateway address is provided by the DHCP server and the value assigned to DefGateway is ignored.

```
ui32 DefGateway = htonl(0xC0A80101);    /* default gateway */
```

`DomainName` - Used to define the domain name of the TargetTCP node, if a domain has been defined. If DomainName is not NULL, the TargetTCP DNS client uses it to resolve IP address lookup requests.

```
char *DomainName;                       /* our domain name */
```

`HostsTable` - Used by the TargetTCP DNS client to satisfy IP address and name requests locally, if possible, before contacting the DNS server. Also, used by the TargetTCP RARP server to translate MAC addresses into IP addresses.

```
HostDesc HostsTable[] =
{ /*lint -e{784} */
  /* name, ip_addr, hw_type, hw_addr */
  {"hsb360", 0xC0A80113, ETH_IFTYPE, "\x08\x00\x3E\x28\x79\xED"},
```

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

```
  {0,0,0,{0}} /* last entry */
};
```

`netRoutes` - Used to specify a list of static routes that, if the list is not empty, are added to the TargetTCP routing table during initialization.

```
ui32 netRoutes[] =
{
  /* Gateway, Destination Network, IP Mask */
  0,    /* end of list */
};
```

`Secrets` - Specifies a list of username, password, home directory, user ID, and group ID entries. Used by the login support for FTP, PPP, and Telnet.

```
Secret Secrets[] =
{
  /* user, password, home, uid, gid */
  {"blunk", "micro", "/rfs", 0, 0 },
  {"snmp", "setokay", "/rfs", 0, 0 },
  {0, 0, 0, 0, 0} /* last entry */
};
```

Other definitions in 'config.c' are required for specific TargetTCP options (IPv6, SNMP, Telnet, etc.) as marked.

File System Use

A file system is required if the TargetTCP FTP or TFTP servers are used. Any POSIX compatible file system can be used. If a Blunk file system is used, other support functions and variables are needed.

For Blunk file systems other than TargetFAT, FsGetId() is called to get the running task's user and group IDs whenever a file's access permissions are checked. It is not used for TargetFAT because the FAT format does not support file access protections.

Also, Blunk file systems support per-task current working directories for resolving relative path names. When PER_TASK_CWD is TRUE in 'config.h', the following definitions are needed for the implementation of this support:

```
void FsSaveCWD(ui32 w1, ui32 w2);
void FsReadCWD(ui32 *w1, ui32 *w2);
void FsResetCWD(ui32 old_w1, ui32 new_w1, ui32 new_w2);
void FsUpdateCWD(ui32 old_w1, ui32 old_w2, ui32 new_w1, ui32 new_w2);
```

These routines are documented in the Blunk file system user manuals and example porting implementations are provided in the directory 'extos'.

OsSecCount is used by Blunk file systems to update file access and modification times. Its type is "volatile unsigned long". If the target system maintains time-of-day, OsSecCount should be

initialized using **mktime()** and thereafter incremented once a second. If this is not done, the access and modification times returned by **stat()** should be ignored.

A final consideration when using Blunk file systems in an environment outside of TargetOS is that the Blunk Standard C and POSIX API names may already be defined by your environment, resulting in linker collision. Name mangling may be needed to address this. More information is provided in the porting instructions for the various Blunk file systems.

Other Considerations

TargetTCP relies on the presence of some system level 'glue' routines that are normally provided by TargetOS. These are provided in files in the 'extos' directory, which also includes examples for the other functions described above.

The symbol BIG_ENDIAN in "sockets.h" needs to be defined or undefined, according to whether the host processor is big or little endian, respectively.

The symbol USING_TARGETOS in the configuration file "config.h" should be set to FALSE. This removes many TargetOS dependencies.

Also, the symbol NVRAM_INC in "config.h" should be set to FALSE, at least initially, to remove dependencies on TargetOS's NVRAM configuration mechanism. If desired, support for accessing non-volatile configuration parameters can be added later. This requires adding support for NvSave().

Unless you are porting to a CPU that Blunk Microsystems has provide assembly language code for the Internet checksum routine, you must set the ASM_CHECKSUM definition in "netutils.c" to FALSE, in order to use the C routine included in that file.

SysPassword() is used by the FTP server, Telnet server, and PPP modules to check if a supplied name is an authorized user and to match user names to their associated password. The example implementation uses the 'Secrets' table configured by the application at compile-time. SysPassword may be modified to use information read from a file or retrieved from non-volatile storage.

If the Telnet server is enabled in 'config.h', you must add support for the taskGetReg() and taskSetReg() functions. These are used to read and save, respectively, a telnet session control block pointer in the task control block of the current running task. These pointers are used to redirect each Telnet server task's stdio to its TCP socket connection.

TargetTCP calls bzero() to zero memory. It is a common function, but some environments don't support bzero(). If yours doesn't, you can make a substitution using the following definition:

```
#define bzero(addr, len)    memset(addr, 0, len)
```

The "assert.h" file shipped with TargetTCP calls the function AssertError() if an assert fails. You can either use your environment's "assert.h" file or provide an implementation of AssertError(). The prototype is:

6576 Leyland Park Dr • San Jose, CA 95120 • Phone 408-323-1758 • Fax 408-323-1757 • www.blunkmicro.com

```
void AssertError(int line, char *file);
```

The TargetTCP FTP server and the Blunk file systems use strcasecmp() and strncasecmp() to perform case insensitive string comparisons. These functions are commonly provided by C compilers, but if your system doesn't support them, you can add implementations to your support code.

The TargetTCP Telnet server and FTP client and server use the function vsnprintf(). TargetTCP also uses the function snprintf(). These functions are commonly provided by C compilers, but if your system doesn't support them, you can add implementations to your support code.

TargetTCP uses the ModuleList array for initialization and, if MENU_INCLUDED in 'config.h' is TRUE, run-time configuration. Several optional function callback pointers are listed in the ModuleList array. These include the following:

BspModule() - used to initialize low level drivers
OsModule() - used to initialize the operating system
The modules defined by BSP_NI_DRIVERS in "bsp.h"

The functions in the ModuleList are called by modInit() with the parameter kInitMod. This provides a handy way to initialize TargetTCP and other included modules at startup. During TargetTCP initialization, the ModuleList functions are called with the parameter kInitNi. This provides a convenient way to automatically install TargetTCP drivers at startup.

If any of these pre-defined module functions (BspModule, OsModule, …) are not useful in your system, you can either supply empty dummy routines or delete them from the ModuleList.