Looking to embed Internet technology into your application? Here's how one company's engineers did it

Designing the Internet into DSP Applications

By Rutger van Dalen

Engineers at Windmill Innovations have succeeded in developing the first eXpressDSP-compliant TCP/IP stack. TCP/IP communications can now be integrated into your TMS320 DSP-based application just like any algorithm that implements the rules and guidelines of the TMS320 Algorithm Standard.

Furthermore, the engineers designed the stack, called bf3Net, for seamless integration with the DSP/BIOS real-time kernel and added a number of plug-ins for the Code Composer Studio development environment that further facilitate the algorithm's integration. How they did it, details of the resulting product, and how the product is modified and optimized by the user should shed light on your own development efforts.

TCP/IP Stack Design Flow

Figure 1 shows the design flow that lies at the basis of the integration process. As with other eXpressDSP-compliant algorithms, the operational behavior and the resulting system resource consumption of bf3Net (MIPS, memory, and so on) are determined primarily by the values of the instance creation parameters. Thus, in essence, the integration process is a quest for that set of parameter values that delivers the right balance between communications performance and resource consumption. This "sweet spot" is unique for each DSP application and so must be found anew in each individual project.

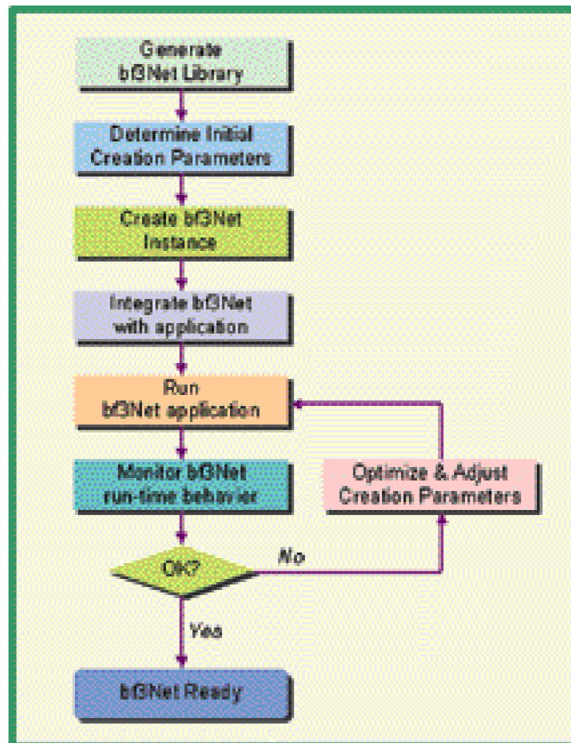

Figure 1. The bf3Net design flow is an iterative process. Starting with an initial set of configuration parameters, the bf3Net stack is optimized by monitoring the run-time behavior.

The design and integration process starts with the creation of the library. To suit the needs of your application, you should customize the library to include only the required protocol modules. That way, the algorithm doesn't burden the DSP system consuming memory and CPU resources unnecessarily. The library creation process is automated by a Code Composer Studio plug-in, in which check boxes are activated for each included protocol; a "Create library" button takes care of the rest.

Another Code Composer Studio plug-in sets the instance creation parameters. An initial set of values—at best an educated guess—serves as the starting point for the integration process. Besides the default creation parameters, you can create an arbitrary number of parameter sets. A built-in code generator creates the resulting header and source files at the click of a button. So far, so good.

The bf3Net Architecture

To understand the creation and integration of the bf3Net protocol stack, look at the algorithm's architecture, shown in Figure 2. At the core, a protocol engine coordinates and synchronizes the operation of the protocol modules, as well as the interaction among them. The engine also manages the data and the internal timers.

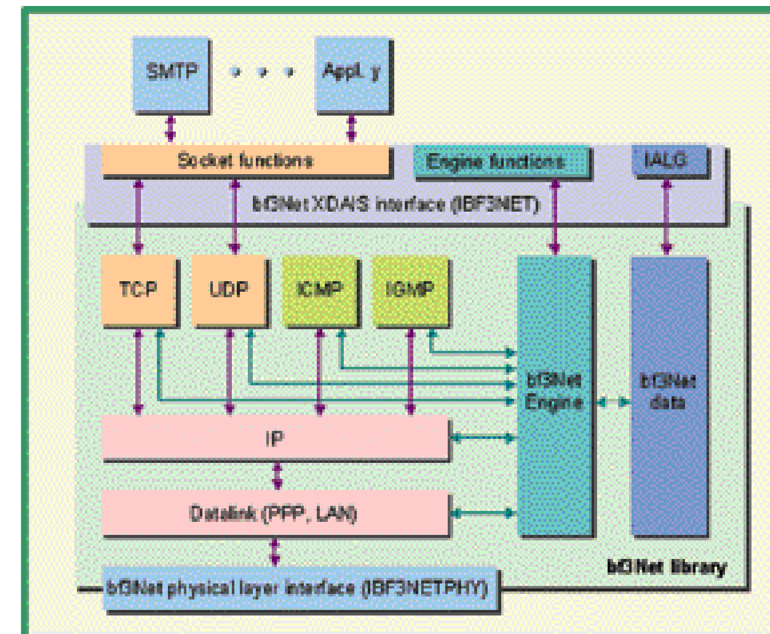

Figure 2. As the central element of the bf3Net architecture, the bf3Net engine manages access to the internal data and to the protocol modules. The data is provided by the DSP application through the IALG interface.

The stack runs as a single thread, which cycles the engine. Consequently, the software doesn't occupy a large number of tasks and also doesn't require intertask communications and synchronization. In addition to a trigger for the engine cycle, the stack requires a timer tick as clock reference for the internal timers. For both those purposes, a real-time operating system or kernel, like DSP/BIOS, is ideally suited, and the application should be designed to directly integrate with such a real-time kernel.

The protocol stack provides two abstract interfaces to the outside world, TMS320 Algorithm Standard and the interface to the physical communications layer (for example, a modem or network interface card). Roughly and for the sake of this discussion, their functions can be divided into three groups: socket functions, engine functions, and IALG functions.

The socket functions serve as interfaces with application protocols (SMTP, HTTP, and the like). The engine functions are called to cycle the engine and increment the timer. Lastly, the IALG interface functions define a "protocol" between the application and the algorithm for creating and deleting algorithm instance objects, allowing multiple instances of an algorithm to be launched in an application (multichannel modem systems, for instance).

The focus here is on the most common case of a single application instance, more specifically for a bf3Net TCP/IP stack making use of the Point-to-Point Protocol (PPP) as the data-link communications layer and a standard AT modem for the physical layer.

Instance Creation and Integration

You create an instance by declaring a handle to the algorithm instance and calling the ALG_create() function.

This single declaration and function call are all that's required to launch an instance of bf3Net, but a few more steps are required to get the TCP/IP communications up and running. First, you must link the instance to the physical communications layer. Next, you must establish the interface between the instance and application protocols. Finally, you must integrate the instance with the system base, in this case DSP/BIOS.

Although the bf3Net library is specific for a data link layer (here PPP), the interface with the physical layer is identical for each implementation. Windmill Innovations' designers have defined the IBF3NETPHY physical layer interface as a uniform interface for physical layer drivers. It specifies the format of the physical-layer function table that must be implemented by each

bf3Net-compatible physical layer driver, addressed with an IBF3NET-PHY_Handle instance handle.

In the case of bf3Net, the instance implements the IBF3NET-PHY_NetFxns function table used as entry point by the physical layer driver. The link between an instance of bf3Net and a physical layer driver is established with a so-called physical layer descriptor, defined as follows:

```
typedef struct IBF3NETPHY_PhyDesc
{
    IBF3NETPHY_PhyFxns      *phyFxns;
    IBF3NETPHY_Handle       phyHandle;
    IBF3NETPHY_NetFxns      *netFxns;
    XDAS_Void               *netHandle;
}IBF3NETPHY_PhyDesc;
```

Therefore you must initialize the physical layer descriptor thus:

```
IBF3NETPHY_PhyDesc phyDesc;
IBF3NETPHY_Handle  phyHandle;
/* initialize <phyHandle> (see remark below)
   */
phyDesc.phyFxns  =&BF3NETPHY_WI_
    ATMODEM_UART16550_IBF3NETPHY;
phyDesc.phyHandle   = phyHandle;
phyDesc.netFxns   = &BF3NETPPP_WI_
    IBF3NETPHY;
phyDesc.netHandle = netHandle;
```

The handle to the physical layer driver, phyHandle, is initialized with functions of the IBF3NETPHY_PhyFxns function table in a way that strongly resembles the ALG_create() method. (Details of this procedure are beyond the scope of this article.)

The instance is linked to the physical layer driver with the following function call:

```
netHandle->fxns->setPhy(netHandle,
    &phyDesc);
```

The integration of the application instance with the physical communications layer is now complete.

Windmill's engineers defined the IBF3NETAPP interface so that the stack can be integrated, and can interact, with application protocols. Similar to IBF3NETPHY, this abstract interface specifies the IBF3NETAPP_Fxns function table, which must be implemented by each bf3Net-compatible application protocol as an entry point for the bf3Net instance. You address the instance of the application protocol with an IBF3NETAPP_Handle handle. The link between an instance of bf3Net and an application protocol instance is made with an application protocol descriptor, defined as follows:

```
typedef struct IBF3NETAPP_ProtDesc {

  struct IBF3NETAPP_Fxns      *protFxns;
       IBF3NETAPP_Handle      protHandle;
       XDAS_Void              *netHandle;

} IBF3NETAPP_ProtDesc;
```

The application protocol instance handle and descriptor are declared and initialized as:

```
IBF3NETAPP_Handle   applProtocol;
IBF3NETAPP_ProtDesc protDesc;
/* initialize <protDesc> (see remark below)
   */
protDesc.protFxns     = &BF3NETAPP_WI_
    APPLPROT_IBF3NETAPP;
protDesc.protHandle  = applProtocol;
protDesc.netHandle   = netHandle;
```

Here, the BF3NETAPP_WI_APPLPROT_IBF3NETAPP constant is the address of the IBF3NETAPP_Fxns implementation of the application protocol. The protocol descriptor is used whenever a bf3Net socket is opened (see below).

The last step concerns the integration of the bf3Net instance with the system base. Although the bf3Net algorithm operates independently of a specific platform, it's most easily integrated with the DSP/BIOS real-time kernel. Only three DSP/BIOS objects are required to run the bf3Net instance: a TSK task object to cycle the bf3Net engine, a PRD periodic function object serving as a timer reference, and a SEM object to wake the task up. For the sake of this discussion, the three objects are named BF3NET_TASK, BF3NET_TIMER, and BF3NET_SEM.

The BF3NET_TASK object is associated with a function, called BF3NET_task(), that cycles the bf3Net engine whenever there is work to do.

The BF3NET_TASK object cycles the bf3Net engine whenever data needs to be processed or actions need to be taken. The bf3Net instance signals DSP/BIOS that that's the case by posting the

#1 In Embedded Communications Software
Serving Industry Leaders Since 1992
Modem & Modem Relay
ADSL, V.92, V.90
V.34, V.32bis, V.32
V.22bis, V.22, V.23
V.21, Bell Modems
Modem Relay
Fax & Fax Relay
V.34, V.17, V.29
V.21 Ch. 2, V.27ter
T.30, T.37, T.38
Fax Relay
VoIP & Speech
G.711, G.723.1,
G.729A,
G.729AB
G.729, G.726
G.722, VoIP
Telephony
LEC (G.165/G.168)
AEC, AGC, CP
DTMF, C-ID I & II
VAD & CNG
DSPs, Microprocessors, RTOS & Applications Supported
GAO Research Inc. is recognized as the world's number one provider of embedded communications software for voice band modems, broadband DSL modems, fax, fax relay, fax over IP, voice, telephony, voice over IP, and
Available for TMS320C5000 & TMS320C6000 families of DSPs
Integrates with MP3 • JPEG • MPEG • TCP/IP • Most popular RTOS and Web browsers
GAO Research Inc.
info@gaoresearch.com
www.gaoresearch.com
416 • 292 • 0038

new
bf3Scp: The ultimate solution for embedded serial communications
1 bf3Scp is a serial communications architecture for full-duplex communication between a PC and a DSP.
2 bf3Scp enables the rapid development of Windows™ user-interfaces for DSP applications.
3 bf3Scp considerably reduces the development time for DSP applications which require serial communications.
Contact us at:
www.windmill-innovations.com
Windmill Innovations
www.windmill-innovations.com

¡Olé!
Tough DSP Jobs are No Match for The Matador Series
Toro A/D: 16 Non-Muxed Channels - 16-bit - 250kHz D/A: 16 Channels - 16-bit - 250kHz
Conejo A/D: 4 Non-Muxed Channels - 14-bit - 10MHz A/D D/A: 4 Channels - 16-bit - 10MHz
Delfin A/D: 32 Non-Muxed Channels - 24-bit - 192kHz A/D D/A: 6 Channels - 24-bit - 192kHz
Vista Video: 4 Channels - NTSC, PAL Video I/O Audio: 2 Channels - 18-bit - 48kHz Audio I/O
Toro Conejo Vista Delfin
Matador Series
Features of the Matador Series of DSP Boards
Texas Instruments' TMS320C6711 Digital Signal Processor
64-bit/32-bit, 3.3V/5V, 33MHz PCI interface
Ultra-flexible trigger modes with hardware event log built in logic
SyncLink/ClockLink interface for easy multi-board synchronization
DSP/BIOS device-drivers for board peripherals
C++ class libraries to accelerate code development
Innovative Integration
... real time solutions!
805.520.3300 phone • www.innovative-dsp.com

BF3NET_SEM semaphore in a rescheduler call-back function that has been registered with the bf3Net instance by application through a IBF3NET function:

```
/* rescheduler call-back function prototype */
XDAS_Void BF3NETPPP_WI_engine
    Reschedule(const IBF3NET_Handle
    handle, XDAS_UInt16 netId);

/* register rescheduler callback function with
bf3Net instance */netHandle->fxns-
>setRescheduler(netHandle,0,BF3NETPPP_
WI_engineReschedule);
/* rescheduler call-back function implementa-
tion */
XDAS_VoidBF3NETPPP_WI_engineResche
dule(const IBF3NET_Handle handle,
XDAS_UInt16 netId)
{
  /* post the semaphore to wake up the task
*/
   SEM_post(&BF3NET_SEM);
}
```

This simple mechanism cycles the bf3Net engine only when required, thus reducing the system overhead to an absolute minimum. Finally, the bf3Net instance requires a timer reference for the management of the internal timers. The PRD object, BF3NET_TIMER, associated with a function called BF3NET_timer(), in turn calls the timerCycle() function of the IBF3NET interface.

The integration of the bf3Net instance with the application is now done.

Establishing Communications

You establish a communications session with bf3Net by opening a socket with the socketOpen() function of the IBF3NET interface. The function takes a pointer to an IBF3NET_SocketCB socket creation block structure, which contains the session-specific communication details, as a parameter. One of the members of the IBF3NET_SocketCB data structure

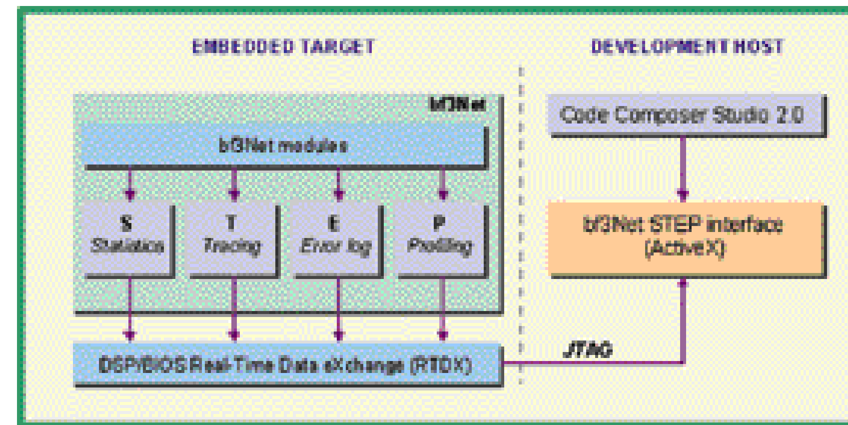

Figure 3: The bf3STEP technology is the basis of the bf3Net integrated debugging strategy. The statistics (S), event-tracing (T), error-logging (E), and profiling (P) information is conveyed to a Code Composer Studio plug-in with TI's Real-Time Data Exchange (RTDX).

is a pointer to the IBF3NETAPP_Fxns function table implementation of the application protocol.

When a bf3Net socket is opened, the underlying communications layers are first established. With a PPP data link, for instance, the modem connection is opened if the bf3Net instance was previously idle. If TCP was selected as the transport layer, the TCP connection will be synchronized with the communications peer.

When all layers have been opened, the application protocol receives an IBF3NET_SOCKETEVENT_OPENED event through the protNotify() function of the IBF3NETAPP_Fxns function table implementation of the application protocol. The protNotify() function must catch and process all events fired by the bf3Net interface, such as the reception and completed transmission of data.

Normally, the socket opening procedure is handled by the application protocol, invisible to the embedded application. You'll gain a better understanding of the bf3Net socket API, however, by following the steps. The procedure starts by declaring and initializing a socket creation block instance.

When that's been done, the bf3Net socket can be opened as follows:

```
/* open the bf3Net socket */
socketID = netHandle->fxns->socketOpen
   (netHandle, &socketCB);
```

The value returned by the socketOpen() function and stored in the socketID variable uniquely identifies the bf3Net socket. You must use this value in all subsequent calls of the IBF3NET socket layer functions.

This simple mechanism cycles the bf3Net engine only when required, reducing system overhead to a minimum.

For example, when the application protocol has received the IBF3NET_SOCKETEVENT_OPENED socket event, the following function call initiates the download of the default file from the HTTP server at IP address 10.0.0.2:

```
/* HTTP file request string */
char httpGetBuffer[33] = "GET http://10.0.0.2/
  HTTP/1.0\r\n\r\n";

/* send the file request string to the Web
  server */
netHandle->fxns->socketWrite(netHandle,
  socketID, (XDAS_UInt8*) httpGetBuffer, 33);
```

To copy the incoming file data to a local buffer, you use the following function call whenever the IBF3NET_SOCKETEVENT_RXDATA_RECEIVED event is fired by the bf3Net instance:

```
/* buffer for received data */
XDAS_UInt8  httpRxBuf[820];

/* length of the received buffer */
XDAS_UInt16 httpRxLength;

/* initialize the length of the receive buffer */
httpRxLength = 820;

/* read the received data from the bf3Net
TCP/IP stack */
netHandle->fxns->socketRead(netHandle,
  socketID, httpRxBuf, &httpRxLength);
```

The httpRxLength variable will contain the number of actually received bytes when the socketRead() function returns.

Optimizing the bf3Net Configuration

Having completed the integration of bf3Net, it's now time to run the bf3Net-based application and optimize the bf3Net configuration. The bf3Net software provides a sophisticated debugging and profiling strategy, based on DSP/BIOS Real-Time Data eXchange (RTDX) for communications with the development host (Code Composer Studio 2.0). A graphical interface plug-in visualizes the data from the target conveyed to the host through RTDX. Four modules implement the bf3Net debugging and profiling strategy: statistics (S), event tracing (T), error logging (E), and profiling (P), or STEP for short.

The *statistics module* gives performance information, e.g., transferred bytes, throughput, checksum errors.

Event tracing reports on connections, negotiations and so on.

Error logging reports errors such as buffer overflows, timer shortages and the like.

The *profiling module* tracks buffer and timer usage to help eliminate stack redundancy,

With the STEP modules leveraging the resources of eXpressDSP, bf3Net provides powerful tools to aid users in developing optimally functioning embedded Internet applications. The optimization procedure is simple: take the current configuration as the baseline, run the application, and monitor the behavior to detect configuration conflicts and errors, then run the application again until you reach the right balance. If you start with an oversized configuration as a "trouble-free philosophy," you can quickly detect the actual resource consumption of bf3Net in your application and arrive at a good solution in only a few iterations. ◆

Rutger van Dalen (rvdalen@windmill-systems.com) is the vice president of engineering at Windmill Innovations BV in Nijkerk, The Netherlands. He has eight years' experience in DSP and embedded application design and was actively involved in the architectural definition and design of bf3Net. Previously, he worked in product development at Lucent Technologies' Switching and Access Systems business unit.

new
www.windmill-innovations.com
bf3Net: The ultimate solution for DSP Internet Connectivity
1 bf3Net is the world's only eXpressDSP-compliant TCP/IP protocol stack.
2 bf3Net offers unsurpassed ease-of-integration.
3 bf3Net strongly leverages Texas Instruments' eXpressDSP™ Software Technology.
Contact us at:
www.windmill-innovations.com
Windmill Innovations
DSP

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2002, Texas Instruments Incorporated