network access products

ZipWire

Software User Guide

100417C
May 2000

CONEXANT™

Table of Contents

List of Figures

List of Tables

1.0 Introduction

The ZipWire family of single-chip HDSL (High-rate Digital Subscriber Line) transceiver chips (referred to as bit pumps) include the Bt8960, Bt8970, and RS8973 devices. The bit pumps are designed to be controlled by an external microcontroller. The microcontroller sets and controls all bit pump internal operation modes during activation, monitors the bit pump's performance during normal operation, and implements steady-state activities such as microinterruptions recovery and temperature variations on the Digital Subscriber Line (DSL).

1.1 Scope

This document describes the software for the Bt8960, Bt8970, and RS8973 bit pumps. This document supersedes the *Bt8960 Software User's Guide* and the *Bt8970 Software User's Guide*.

1.2 System Overview

The ZipWire software implements all necessary bit pump control and monitoring operations, including activation process, performance monitoring, and test modes. This software package is an integral part of the bit pump and must be used without any significant changes to guarantee correct operation and optimal bit pump performance.

The software is supplied in a C source code format targeted to the Intel 80C51 controller family. The source code can be used in a standalone configuration consisting of a single microcontroller or in a dual processor configuration consisting of a second host processor that runs the user code and communicates with the microcontroller through an RS-232 channel. The second host processor is typically a PC. The ZipWire software is targeted to the 80C32 microcontroller, which is a member of the Intel 80C51 family.

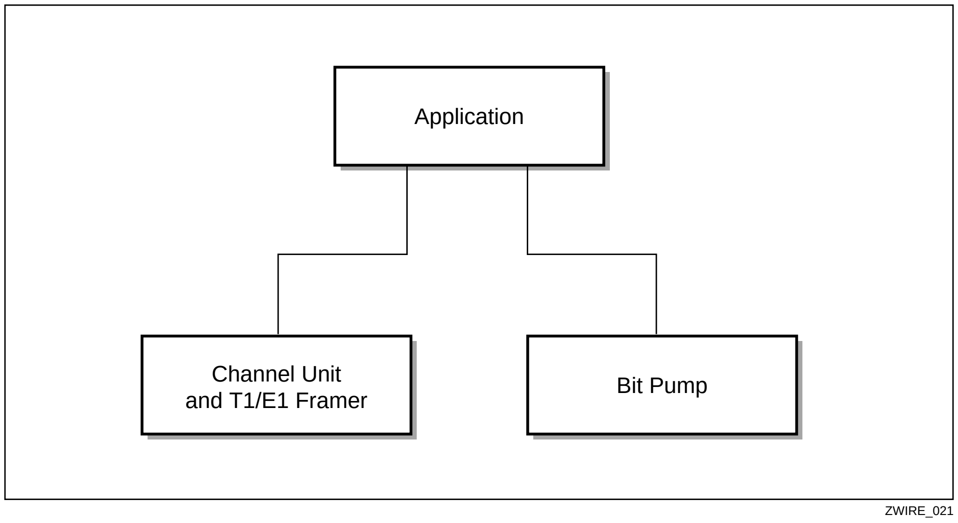
The ZipWire software is targeted to the 80C32 and the hardware specifics of the ZipWire EValuation Module (EVM) such as port interfaces, address map, interrupt lines, etc. If a non-80C32 processor is used, or if an 80C32-based system implements a different hardware design than the EVM, the C source code must be modified and recompiled.

A major design goal for the ZipWire software is portability and ease of conversion for different hardware and compiler environments. However, this cannot completely eliminate the need to tune the software for specific implementations because of the hardware-dependent nature of the code. This manual includes detailed instructions on modifying the ZipWire software.

The application code interacts with the bit pump via Application Programming Interface (API) commands, which can be called either at the code level (for applications integrating the ZipWire software in the single processor configuration) or by using the RS-232 serial channel (in the dual processor configuration). The API comments allows access to all statuses required to implement an HDSL system, including noise margin, pulse attenuation, Loss Of Signal (LOS), etc.

The ZipWire software is conceptually partitioned into three sections of code: application code, channel unit code, and bit pump code as illustrated in Figure 1-1. The application code supports the bit pump device with or without an HDSL channel unit device (RS8953B, RS8953SPB, or Bt8954) and T1/E1 framer device (e.g., Bt8370). The application code discussed in this manual assumes there is no channel unit device in the system.

Figure 1-1. ZipWire Software Hierarchy



The channel unit code is discussed in the *RS8953B Application and Channel Unit Software Developer's Guide* (NCHUDG1B). The application code discussed in NCHUDG1B assumes there is a channel unit device in the system.

The application code is inevitably the responsibility of the end user. In the EVM, Conexant provides an example of this application code. To obtain an EVM, contact your local sales representative. The EVM is available with or without a line card containing a channel unit device.

The ZipWire software handles the complete activation process with no need for application intervention. All bit pumps can go through activation in parallel, with practically no degradation of activation time as listed in Table 6-3.

Although the control software does not perform any Digital Signal Processing (DSP) algorithms, some sections of the code involve real-time operations, especially during activation. Applications that integrate the ZipWire software should take this into account and follow the guidance given in Chapter 6.0.

1.3 Contents of This Manual

This manual contains all information required to successfully modify, compile, and use the ZipWire software. Please read this manual prior to writing application code for the bit pump. Table 1-1 lists the chapters in this manual.

Table 1-1. Summary of Contents

Chapter	Contents
Software Overview	Presents an overview of the software design principles, portability considerations, hardware-related issues, application interface principles, and application code integration constraints
Application Software	Explains the sample application code provided with the ZipWire software
Bit Pump Software Operation	Describes the bit pump code structure, main functions, file names, and reserved names
Compiling the Bit Pump Source Code	Explains how to modify and compile the C source code to match the application hardware and software environment
Real-Time Constraints	Details the timing constraints of the bit pump software
API Overview and Serial Communications Interface	Details the user interface operation, command structure, and status responses
Diagnostic Tests	Explains diagnostic tests to verify correct software and hardware operation
Appendices	A. API Command Set Reference B. Calibrating Noise Margin Table C. DIP Switch Settings D. Definition of DIP_SW Variable E. References

The software is described in a top-down fashion in Chapters 2.0–4.0.

The bit pump is controlled using API commands. Use the detailed information in Chapter 3.0 when writing the application software. It outlines the required initialization and parameter setting sequences, accompanied by the ZipWire EVM application example.

Chapter 4.0 provides an overview of the bit pump software. The bit pump software does not require modification, but a basic understanding is useful when integrating the ZipWire software or debugging problems.

Use Chapter 5.0 when modifying and compiling the source code, and Chapter 6.0 when integrating the customer's application software with the ZipWire software. Chapter 7.0 contains information on how to use the ZipWire software and API command set in a transceiver application.

Appendix A: is a reference for the API command set.

1.4 Conventions Used in This Manual

Tables 1-2 and 1-3 list software and document naming conventions, respectively.

Table 1-2. Software Naming Conventions

Software Item	Software Convention	Document Examples
macro	ALL_CAPITALS	INIT_BP_PTR, TIMER_BREAK(t3)
compiler directive	ALL_CAPITALS	TDEBUG, TEMP_ENV
API parameter	_ALL_CAPITALS	_PRESENT, _BIT_PUMP0
function	*_Initial Caps ()*	*_BtInitialize()*
global variable	*_lower_case*	*_bp_vars[], _int_reg[]*
local variable, data structure	*lower_case*	*user_setup_high, transceiver*
pointer	**lower_case*	**bp_ptr, *bp_mode_ptr*

Table 1-3. Document Naming Conventions

Item	Convention	Document Examples
register name	lower_case (hex address)	global_modes (0x00)
file name	lower.case	btint.c
code section	lower.case.n	user.c.3
colloquialism	“lower case”	“one-shot,” “in-line”

A software macro is a software block that the C precompiler substitutes “in-line” in the software in one or more places. A macro runs faster than a software function, but requires more ROM because the same code is duplicated in one or more locations. The ZipWire software uses macros to expedite accesses to the bit pump registers. The format for a macro is:

```
#define macro macro_body
```

A compiler directive (also called a preprocessor directive) conditionally compiles source code. The format for a compiler directive is:

```
#ifdef directive
```

1.5 Terminology

1.5.1 Digital Subscriber Line (DSL)

In this manual, DSL refers to the twisted pair of wires in such contexts as DSL conditions or DSL rate.

1.5.2 HDSL Terminal Unit—Remote (HTU-R)

The HTU-R is also referred to as the Network Terminal Unit (NTU) or the Customer Premise Equipment (CPE). This manual uses the term HTU-R exclusively.

1.5.3 HDSL Terminal Unit—Central Office (HTU-C)

The HTU-C is also referred to as the Line Terminal Unit (LTU) or the Central Office (CO). This manual uses the term HTU-C exclusively.

1.5.4 Far-End

The far-end is the terminal unit on the opposite end of the DSL. For example, if the far-end is an HTU-R, the near-end is an HTU-C.

1.5.5 Near-End

The near-end is a terminal unit which is either an HTU-R or HTU-C. For example, if the near-end is an HTU-R, the far-end is an HTU-C.

1.5.6 Data Rate or Bit Rate

The data rate or bit rate is the rate of data received on the RDAT output from the bit pump or the rate of data transmitted on the TDAT input to the bit pump. The data rate can be computed as follows:

$$\text{data rate} = (\log_2 \text{signal levels}) \times \text{symbol rate} = 2 \times \text{symbol rate}$$

HDSL uses 2B1Q (2 Binary to 1 Quaternary) encoding which has four signal levels. The BLCK output from the bit pump runs at the data rate.

1.5.7 Symbol Rate or Baud Rate

The symbol rate or baud rate is the rate of transmission over the DSL. The symbol rate is one-half the data rate. The QCLK output from the bit pump runs at the symbol rate.

1.5.8 Symbol Time

Symbol time is the transmission time of one 2B1Q symbol over the DSL. It is computed as follows:

$$1 \text{ symbol time (ms)} = (2 \text{ bits} \,/\, \text{symbol}) \times (1 \,/\, \text{data rate in Kbps})$$

For example, the symbol time at a data rate of 1,168 Kbps is

$$1 \text{ symbol time (ms)} = (2 \text{ bits} \,/\, \text{symbol}) \times (1 \,/\, 1{,}168 \text{ Kbps}) = 1.71^{-3} \text{ ms}$$

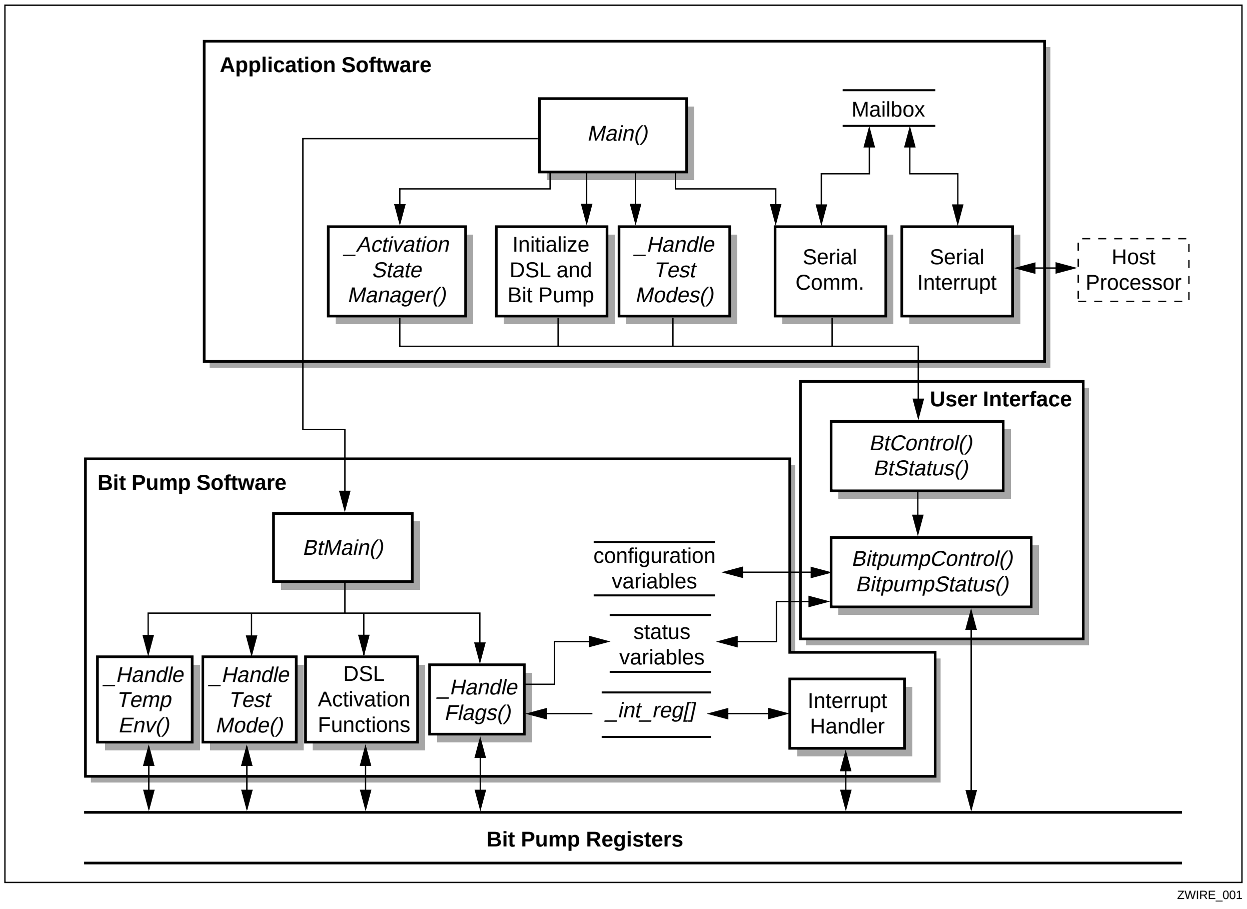
1.5.9 Symmetric Digital Subscriber Line (SDSL)

The SDSL is a derivation of HDSL technology that typically does not use a channel unit device. Several vendors provide proprietary SDSL specifications to ensure interoperability, but currently there is no SDSL standard.

2.0 Software Overview

The ZipWire software handles and monitors the activation and normal operation activities of one or more HDSL bit pumps. Figure 2-1 illustrates a software calling tree for the principal functions.

Figure 2-1. ZipWire Software Calling Tree



As illustrated in Figure 2-1, the ZipWire software includes the following three major functional blocks:

1. Application Software—Contains the application-specific software to initialize the system and control system level tasks.
2. Bit Pump Software—Includes these major components:
   - *_BtMain()*—Activates the bit pump activation state machines for each bit pump present in the system.
   - DSL Activation Functions—Runs the principal bit pump activation functions. These functions proceed through a series of states that control and monitor the activation of the DSL connection. The *_HtucControlProcess()* function is the activation function for the HTU-C; *_HturControlProcess()* is the activation function for the HTU-R.
   - Interrupt Handler—Executes every time one of the bit pumps initiates an interrupt. As illustrated in Figure 2-1, this function reads the irq_source register (0x05) and copies it to a global variable, *_int_reg[]*. The *_HandleFlags()* function then polls this global variable (using the **int_ptr* pointer) to check for status changes. For example, the LOS check in *_HandleFlags()* in file bitpump.c is shown here:

     ```
     if (int_ptr -> bits.irq_source.low_felm)
     ```
3. User Interface—Processes Application Program Interface (API) commands. The interface is implemented either at the code level or via a serial communication protocol. In both options, the API command set is defined by the set of parameters passed to the *_BtControl()* and *_BtStatus()* functions.

The three major functional blocks are detailed in later chapters.

2.1 Software Integration

Details of software integration are discussed throughout this document. This section provides an overview of integrating the ZipWire software with the customer's application software.

The ZipWire software contains the following files and directories in the top-level directory:

- application software files
- bit pump directory
- channel unit (chanunit) directory

The channel unit software is discussed in the *RS8953B Application and Channel Unit Software Developer's Guide* (NCHUDG1B).

It is recommended that customers copy all files in the bit pump directory into their systems and only make the minor modifications discussed in Chapter 5.0. On the other hand, customers must write the application software but can use the ZipWire application software as an example. The ZipWire application software is targeted to the EVM system and is discussed in Chapter 3.0.

2.2 Software Portability

The ZipWire software is designed with code portability as a major goal. To meet this goal, the ZipWire software is written in C, which can be easily modified to match different hardware and software environments.

The code itself has hardware-related parameters which must be defined. For example, the absolute address space allocated to each bit pump depends on the specific hardware design and address decoding details.

Another example of a hardware- and software-dependent code section is the definition of *_BpInterruptHandler()* in btint.c. The function is part of the supplied software and should not be modified, but the function header definition and notation is both processor- and compiler-dependent. These examples demonstrate the kind of modifications required in the source code to successfully compile and run the ZipWire software.

Chapter 4.0 describes the required modifications which are clearly marked and commented upon in the source files.

2.3 Real-Time Constraints

The ZipWire software controls and sequences all activation operations that include real-time operations. In a system where the ZipWire software shares processing resources with the user's application software, care must be taken to not impair bit pump performance. Guidelines for handling real-time operations within applications are explained in detail in Chapter 6.0.

2.4 API Command Set

Because the ZipWire software handles all bit pump operations (activation, performance monitoring, etc.), all user interaction with the bit pumps is performed through the software and not by directly accessing the bit pump. In some cases, performing a read/write operation can impair the proper operation of the control software.

The ZipWire software contains an API command set that allows full control of the bit pumps and performance monitoring. The ZipWire software responds to the API commands by performing the requested action or supplying the requested status.

There are two configurations for transferring commands and parameter status between an application and the ZipWire software:

1. In a single processor architecture, the interface is implemented by directly calling the *_BtControl()* and *_BtStatus()* API functions. The application software issues commands and receives status by calling these functions. Using the API commands makes it easier to incorporate future code changes.
2. In a dual processor architecture, the bit pumps are controlled by an 80C32 processor, and the application interfaces with the bit pumps through the serial communication (UART) channel. In this configuration, the commands and status are sent as messages over the serial channel.

Both configurations use the same set of API commands. The command set is designed for maximum flexibility in operating the bit pumps and supplies all necessary performance and status monitoring information. Each bit pump can be controlled and monitored individually via the API commands.

The API functions decode commands issued by the application and perform the required actions, such as executing the command in the case of a control command, and sending the required information in the case of a status request command. The functions that perform these actions are the two API functions: *_BtControl()*, which performs control commands, and *_BtStatus()*, which performs status request commands.

The ZipWire EVM includes a 80C32 microcontroller that can be configured to work either in the single or dual processor architecture. In dual processor architecture, the application software runs on a separate host processor. The EVM uses a dual processor architecture and runs a User Interface Program (UIP) on a PC to simulate the host processor. For more information on the UIP, see *BtHDSL-EVS User Interface Program (UIP) User's Guide* (UGHDSL_1A).

Chapter 7.0 details the structure of the API commands and the communication protocol for both API configurations. Appendix A is a reference guide to the API commands and status bits with details on command use, syntax, and operation.

2.5 Types and Number of Bit Pumps Supported

The ZipWire software supports the Bt8960, Bt8970, and RS8973 bit pumps. The software uses the GET_BITPUMP_TYPE macro to read the global_modes register (0x00) to identify the particular bit pump.

In a single processor configuration, the software can be easily modified to support a theoretical limit of 256 bit pumps without major changes to the API protocol. The practical limitation is a factor of the host processor bandwidth and DSL startup time requirements, if any. High-end applications that use a PowerPC or similar category processor can start 24 bit pumps in parallel. In these applications, a Real-Time Operating System (RTOS) is recommended to provide a multitasking executive. The baseline ZipWire software supports a maximum of six bit pumps because this is the number used in a 3E1 repeater application. See Section 5.9.5 for more details.

2.6 HTU-C versus HTU-R Distinction

The software can be configured to distinguish between the HTU-R and HTU-C either at compile time or at run time. By defining either the HTUR or the HTUC directive (discussed in Section 5.7), the distinction is enforced at compile time. If both of these directives are defined at compile time, the software must decide on the terminal type at run time. On the EVM, the terminal type is selected by using DIP switch #1 which connects to a microcontroller port.

2.7 HDSL Specifications

The ZipWire software implements the DSL activation procedures per the HDSL specifications listed in Table 2-1.

Table 2-1. HDSL Specifications

Governing Body	Specification Number	Issue Date
ANSI(1)	T1E1.4/94-006 (working draft)	June 6, 1994
ETSI(2)	TS-101-135	February, 1998
ITU(3)	Recommendation G.991.1	October, 1998

NOTE(S):
(1) American National Standards Institute
(2) European Telecommunications Standards Institute
(3) International Telecommunication Union

There are no differences among the specifications regarding the DSL activation requirements. However, the specifications assume that a channel unit device is part of the system design. For bit pump-only applications, the ZipWire software simplifies to the DSL activation procedure. The ZipWire software implementation for the DSL activation procedure is discussed in Sections 3.5–3.10.

2.8 ZipStartup Software

This manual and the software occasionally mention the ZipStartup software. The ZipStartup software greatly reduces the startup times on subsequent startup attempts in which the loop condition has not significantly changed.

The ZipStartup software is an add-on to the ZipWire software and must be purchased separately. The order number is BT50-J001-002. Details about the ZipStartup software are in the ZipStartup documentation.

3.0 Applications Software

Figure 3-1 illustrates a detailed flowchart of the application software for a system with three bit pumps. Table 3-1 defines the functions shown in this flowchart and lists section numbers that provide additional information.

Figure 3-1. Application Software Flowchart

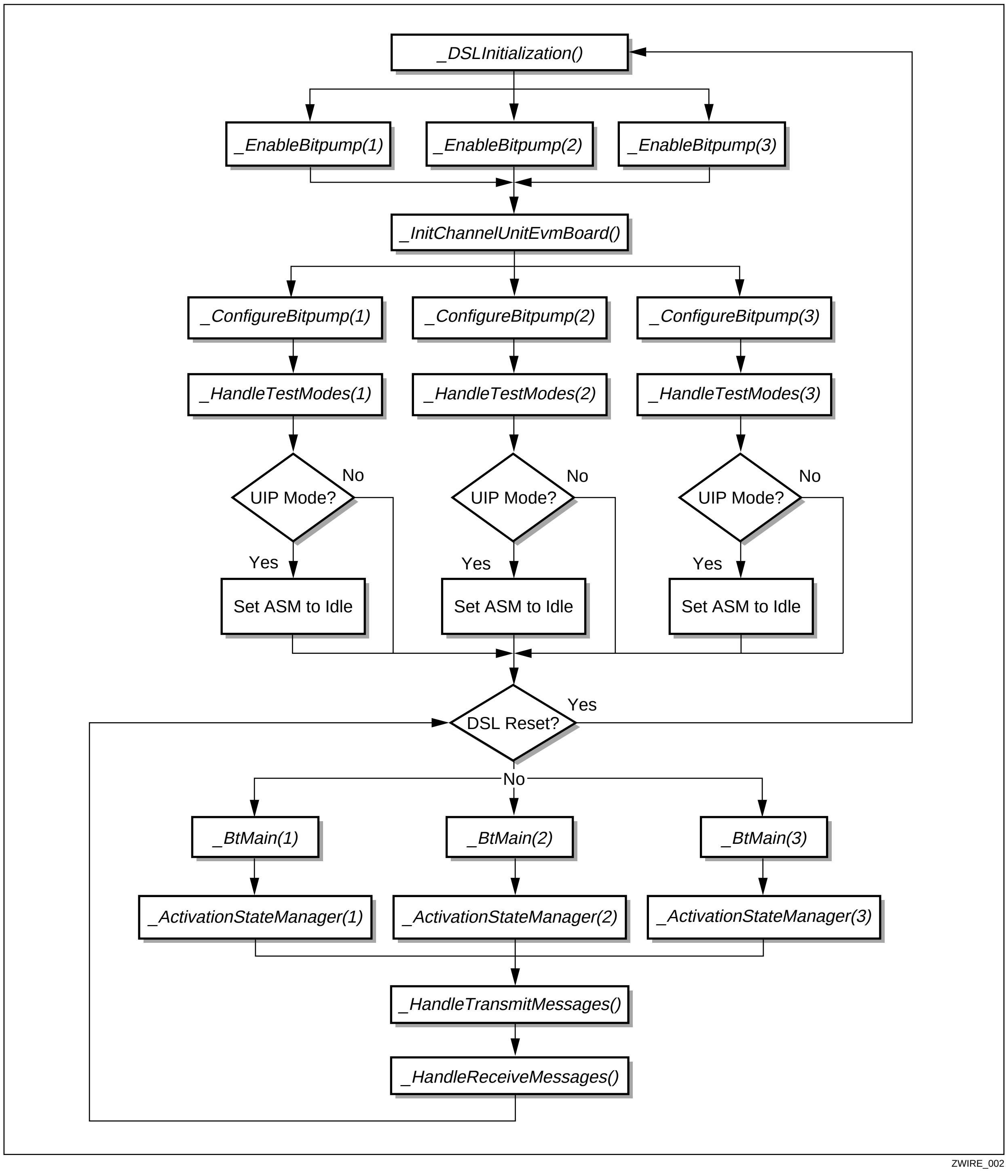


***Table 3-1.* Functions Called by the Application Software**

Function	Description	Section Number
_DSLInitialization()	Bit Pump Initialization	Section 3.1
_InitChannelUnitEvmBoard()	Channel Unit and Framer Initialization	none (function is not applicable)
_EnableBitpump	Bit Pump On and Off Setting	Section 3.2
_ConfigureBitpump()	Bit Pump Configuration (i.e., data rate, scrambler, etc.)	Section 3.3
_HandleTestModes()	Test Mode Processing	Section 3.4
_BtMain()	Bit Pump Main Processing Loop	Section 4.1
_ActivationStateManager()	DSL Activation State Machine	Section 3.5
_HandleTransmitMessages()	Serial Messages Transmitted via RS-232	Section 3.12
_HandleReceiveMessages()	Serial Messages Received via RS-232	Section 3.12

3.1 _DSLInitialization() Function

The *_DSLInitialization()* function is called by the application software on both the HTU-C and HTU-R. When the microcontroller is powered-on or reset, the *_DSLInitialization()* function is called which subsequently calls the functions listed in Table 3-2 that initialize the bit pump software:

Table 3-2.* Functions Called by *_DSLInitialization()

Function or Macro	File Name	Description
_BtSwPowerUp()	btmain.c	Bit pump software initialization. Must be executed prior to any other bit pump-related operation.
_MaskBtHomerInt()	btmain.c	Masks all bit pump interrupts for all present bit pump chips.
_Init8051()	init51.c	Used only in 80C32 environment. Initializes 80C32-specific parameters.
_InitVirtualTimers()	intbug.c	Used only with the INT_BUG (see Section 5.7) directive. Initializes virtual timers.
_InitGenPurposeTimers()	timer.c	Initializes the general-purpose timers. Only needs to be called once at the beginning of the program. See Section 4.5.
_BIT_PUMP_PRESENT API command	api.c	Determines if the bit pump is present by verifying a match of the value written to and read from the bit pump indirect memory.

The application Activation State Machine (ASM), discussed in Sections 3.9 and 3.10, is set to CONFIGURATION_STATE during initialization. If the ASM remains in this state, it starts the activation sequence. If a test mode is selected or the UIP mode is later selected during startup, the ASM state is changed to SYSTEM_IDLE.

3.2 _EnableBitpump() Function

The function *_EnableBitpump()* calls the API command listed in Table 3-3.

Table 3-3. Function Called by _EnableBitpump()

Command Name	Parameter	Remarks
Bit Pump On/Off	_PRESENT	Turn bit pump on

Before another API command is issued to a bit pump, the bit pump must be turned on using the Bit Pump On/Off API command. Following power-up, all bit pumps are declared to be off. All commands (other than the On/Off command) sent to a bit pump that is turned off are ignored by the ZipWire software.

Turning on a bit pump means that the corresponding bit pump is installed in the system. Not all bit pumps installed in the system must be turned on. A bit pump can be turned on or off at any time, and any combination of bit pumps turned on or off is allowed. Turning a bit pump on (even if it is already turned on) resets the bit pump, and sets the user-configurable parameters to their default values.

3.3 _ConfigureBitpump() Function

The function *_ConfigureBitpump()* calls the API commands listed in Table 3-4.

Table 3-4. Functions Called by _ConfigureBitpump()

Command Name	Parameter	Remarks
Symbol Rate [(1)]	On the EVM, the symbol rate is based on the DIP switch #1 setting (see Figures C-1 or C-2).	API value = (Symbol Rate / 4,000) e.g., 98 = 392,000 / 4,000 (784 Kbps data rate)
Terminal Type	_HTUC	Central Office Terminal
	_HTUR	Remote Terminal
Analog AGC Configuration	_SIX_LEVEL_AGC	6-level AGC is implemented
Startup Sequence Source	_INTERNAL	Use internally generated scrambled 1s during activation
Transmit Scrambler	_ACTIVATE_SCR	Use bit pump scrambler
Receive Descrambler	_ACTIVATE_DESCR	Use bit pump descrambler
Data Transfer Format	_SERIAL_SWAP	Sign bit followed by Magnitude bit
Other Side Bit Pump	_BT	Bit pump on other side is a Conexant bit pump
LOST Time Period (only the HTU-C application ASM checks this timer)	40	Set LOST = 4 seconds
Auto Tip/Ring Reversal	ON	Enable automatic tip/ring reversal detection and correction
Activation Time-Out	_ACT_TIME_VARIABLE	Set activation time-out based on the data rate (see Table A-3)
ZipStartup Code	OFF	Disable ZipStartup feature

NOTE(S):
(1) The Bit Pump On/Off command (Table 3-3) and Symbol Rate command must be called first and second, respectively. The remaining API commands (Table 3-4) can be called in any order.

Some API parameters depend on the presence or absence of a channel unit. When a channel unit is not present, the bit pump hardware must:

- Use its scrambler and descrambler to provide a random signal pattern on the DSL.
- Provide the sequence source of 2- and 4-level scrambled data for activation.
- Cooperate with the software to provide tip/ring detection and reversal.

The lack of a channel unit device also implies a startup that is not required to follow the HDSL specifications. In this case, the LOST timer is increased to 4 seconds to provide better synchronization between the HTU-C and the HTU-R. The activation time-out is also changed from a fixed value of 30 seconds per the HDSL specification to a value based on the data rate.

For proper operation of the bit pumps, all configuration and operation parameters must be set to their correct values (depending on the specific application) prior to activating the bit pumps. These parameters include design-specific information (such as analog AGC configuration, data transfer format, activation sequence source, etc.) and application-specific parameters (such as symbol rate, terminal type, etc.). If the default value of an API parameter (as listed in Table 3-5 and Appendix A:) matches the desired option, the API command is not required.

Table 3-5. User Configurable Parameters and Their Default Values

Command	Default Parameter
Bit Pump On/Off	Off
Terminal Type	HTU-C
Other Side Bit Pump	Non-Conexant Bit Pump
Analog AGC Configuration	6 Level
Symbol Rate	784 Kbps
LOST Time Period	1 second
Data Transfer Format	Serial
Startup Sequence Source	External
Transmitter Scrambler On/Bypass	Bypass
Receiver Descrambler On/Bypass	Bypass
Operate Nonlinear EC	Off
Write Transmitter Gain	Calibration Value
Auto Tip/Ring Reversal	Bypass
Activation Time-Out	30 seconds
ZipStartup Configuration	Disabled

3.4 _HandleTestModes() Function

The *_HandleTestModes()* function checks if a test mode is selected on the EVM by reading the DIP switch settings. See Figure C-2 for the test modes available with the DIP switches. When a test mode is activated, the ASM is placed in the SYSTEM_IDLE state that prevents the DSL activation from interfering with the test mode.

The *_HandleTestModes()* function is different from the *_HandleTestMode()* function that is the test mode state machine. The function *_BtMain()* calls *_HandleTestMode()* (see Section 4.1).

3.5 _ActivationStateManager() Function

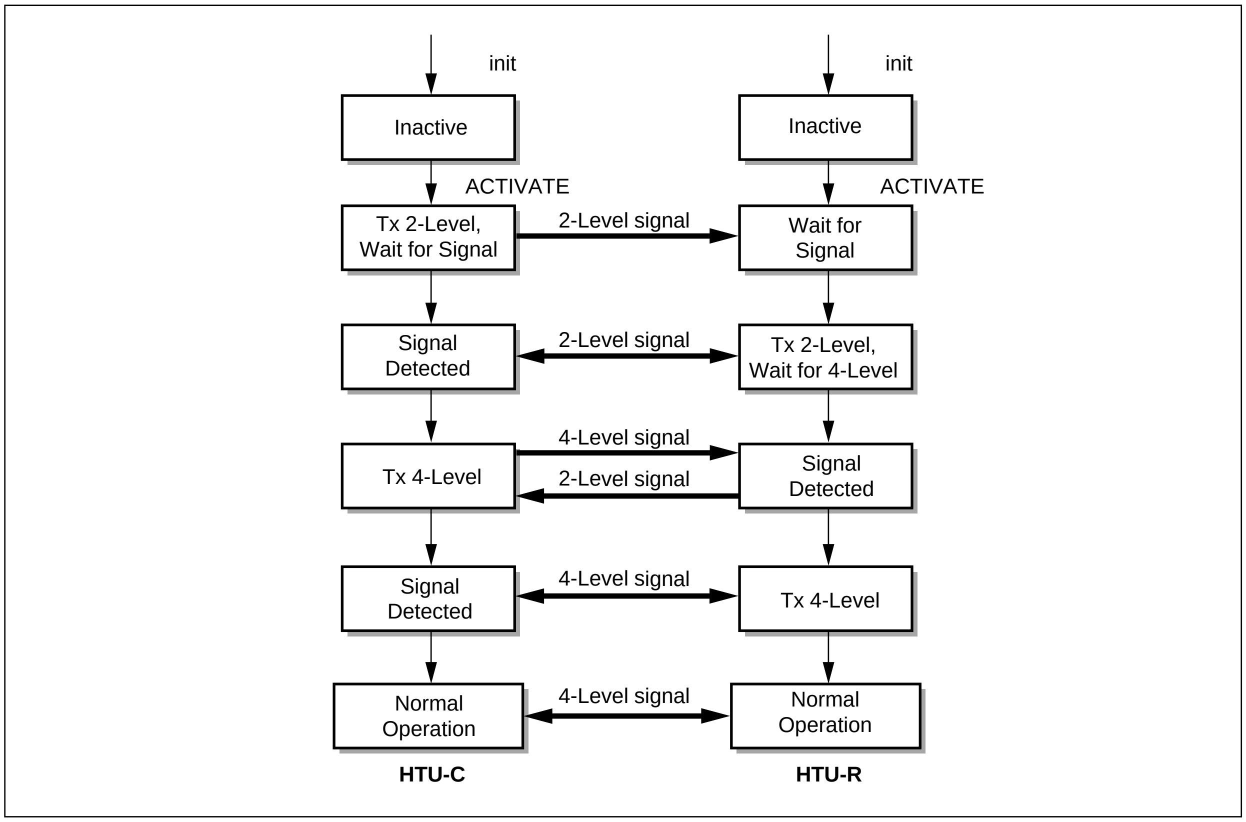
This section describes the recommended Activation State Machine (ASM) when a channel unit device is not part of the HDSL system. The ASM is derived from the ASM in the HDSL specifications (see Section 2.7) by removing the channel unit specifics. When a channel unit is present, see NCHUDG1B for the appropriate ASM. In the ZipWire software, the ASM is implemented in the *_ActivationStateManager()* function in dsl_asm.c.

The ASM needs to issue only four API commands to control the bit pump, assuming the bit pump has been properly configured using the appropriate API commands. The API commands and their functions are as follows:

- _ACTIVATE—Activates the ASM
- _STARTUP_STATUS—Monitors the bit pump activation status, which allows the application code to determine the bit pump activation state and detect any error conditions
- _TRANSMIT_EXT_DATA—Transmits externally supplied payload data
- _DEACTIVATE—Deactivates the bit pump

Figure 3-2 illustrates the activation procedure for the HTU-C and HTU-R. When the bit pump on the HTU-C receives an _ACTIVATE command, it transmits a 2-level signal to the far end. When the bit pump on the HTU-R receives an _ACTIVATE command, it waits for a signal.

Figure 3-2. Bit Pump Activation Procedure



The order in which the _ ACTIVATE commands are issued to the HTU-C and HTU-R is not important, but the complete activation procedure only starts after both terminals are activated. When the HTU-C is activated first, it starts a 2-level transmission but does not receive any response from the HTU-R until the HTU-R is activated. When the HTU-R is activated first, it waits for the detection of an HTU-C signal.

After the HTU-R detects the 2-level signal, it performs frequency lock, DSL characterization, and echo cancellation coefficient calculation. Upon completion, the HTU-R transmits a 2-level signal back to the HTU-C and waits for a 4-level signal. The HTU-C then performs characterization based on the HTU-R 2-level signal. When HTU-C completes its characterization, it sends a 4-level 2B1Q (2 Binary to 1 Quaternary) encoded signal. At this stage, normal operation is reached with transmission of a 2B1Q signal across the DSL.

Operating in parallel to the application-level ASM, the bit pump ASM transitions many states during the bit pump training process. The bit pump ASM software functions are called *_HtucControlProcess()* on the HTU-C and *_HturControlProcess()* on the HTU-R. The application code must repeatedly call the *_BtMain()* function to proceed through the bit pump ASM. The bit pump ASMs are discussed in Section 4.2.

3.6 Bit Pump Startup Status Information

The applications ASM monitors bit pump statuses during activation to decide whether or not the activation has completed successfully. Sections 3.9 and 3.10 explain when specific statuses are monitored by the application ASMs. This section provides an overview of the status information.

Table 3-6 lists the important statuses during activation and describes the statuses together with the API command used to access each status parameter. Table 3-7 shows additional bit pump responses that do not relate directly to the activation procedure. Further information on the API commands is given in Appendix A:.

Table 3-6. Startup Status Parameters

Indication	Description	Command Name
Noise Margin OK	1 = noise margin higher than –5 dB	Bit Pump Status
Loss Of Signal (LOS)	0 = received signal present 1 = no received signal present Status is based on average far-end signal power measurement. This status is valid at all times.	Bit Pump Status
Loss Of Signal Time-out (LOST)	1 = LOS condition present for more than the LOST time period (programmable) The LOST response is valid only after a Deactivate command is issued.	Bit Pump Status
Activation Timer	1 = expiration of activation interval	Bit Pump Status
Transmit 4-Level	1 = bit pump is transmitting a 4-level signal	Bit Pump Status
Tip/Ring Reversal	1 = tip/ring reversal has been set by the bit pump	Bit Pump Status
Normal Operation	1 = bit pump has completed the activation process	Bit Pump Status

Table 3-7. Additional Status Parameters

Indication	Description	Command Name
Cable Attenuation	Total signal attenuation (in dB). FELM can be read once after startup. It is a measure of the DSL attenuation and should have minor variations.	Far-end Signal Attenuation
Noise Margin Reading (NMR)	An estimate of the noise margin reading (in dB) indicating the tolerable increase in noise level while still maintaining a BER < 1×10^{-7}.	Noise Margin

3.7 NMR Reading

The software provides a Noise Margin Reading (NMR) rather than a Signal-to-Noise Ratio (SNR) reading. The NMR is faster to calculate than SNR, and provides an adequate measurement of the DSL quality.

The software supports an API command that provides the instantaneous NMR. The NMR is calibrated to the Gaussian noise model defined in the ETSI specification, which states that 0 dB corresponds to a Bit Error Rate (BER) of 10^{-7}.

Because noise spikes can cause a temporary high NMR, some type of filtering on the NMR value is recommended, i.e., either average 10 individual samples or wait for the NMR to be high for 2 seconds before taking action. The ZipWire software implements the delay by starting a 2-second timer within the PENDING_DEACTIVATE_STATE in the application ASMs.

The threshold is application- and customer-specific. The ASMs check for a NMR > –5 as a pass criterion for startup and as one necessary condition for staying in the ACTIVE_TX_RX_STATE.

3.8 LOS

In bit pump terms, a LOS condition is equivalent to a low FELM alarm. When a bit pump low FELM alarm occurs, the bit pump interrupts the microcontroller. *_BpInterruptHandler()* copies the bit pump irq_source register (0x05) to the *irq_source* global variable in *_int_reg[]*. The *_HandleFlags()* function then polls this global variable (using the **_int_ptr* pointer) to check for status changes.

If the low_felm flag in *irq_source* is set, *_HandleFlags()* sets a flag:

```
status_reg.bits.los = ON
```

The ASMs ensure that this flag is cleared as one of two conditions needed to remain in ACTIVE_TX_RX_STATE (see Sections 3.9.6 and 3.10.6).

3.9 HTU-C Activation

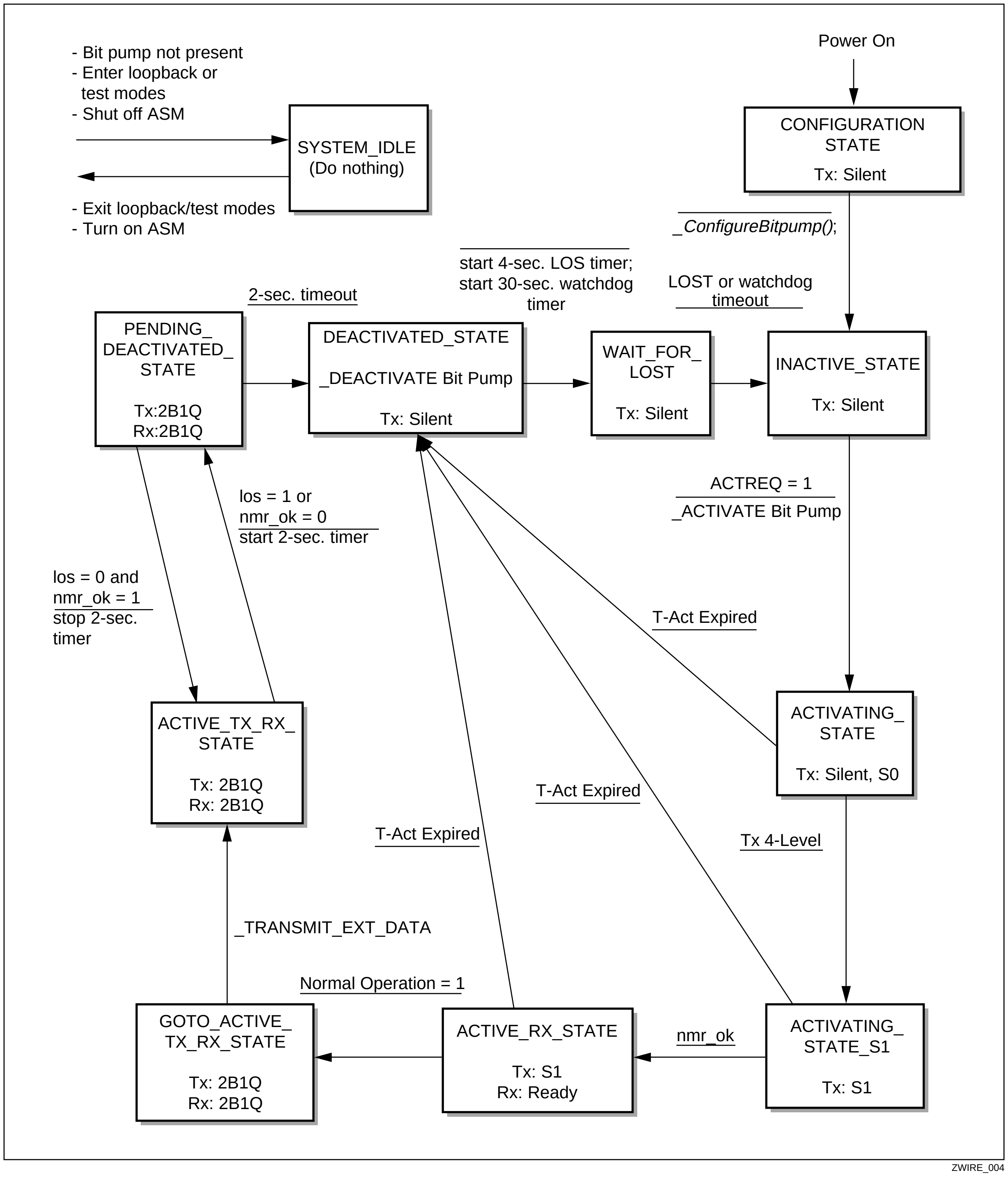
This section discusses processing in the states. Figure 3-3 illustrates the ASM for the HTU-C. State names are capitalized to match the actual software names.

The ASM figures in this manual depict states as boxes or ovals. Lines between states indicate transitions between states. The lines may include a text block with the name of an optional event (above a horizontal line) and an optional action (below a horizontal line).

In the following description, the responsibilities of the ASM software and bit pump code are listed for each state. The ASM accesses the bit pump code through the *_BtControl()* and *_BtStatus()* function calls.

The activation sequence illustrated in Figure 3-2 occurs during the ACTIVATING_STATE and ACTIVATING_STATE_S1 states in Figure 3-3.

Figure 3-3. HTU-C Activation State Machine



3.9.1 CONFIGURATION_STATE

ASM—In CONFIGURATION_STATE, the ASM calls the function *_ConfigureBitpump()* discussed in Section 3.3.

Bit Pump Code—The bit pump code processes the API commands called by *_ConfigureBitpump()*.

3.9.2 INACTIVE_STATE

ASM—In INACTIVE_STATE, the _ACTIVATE API command is issued to initiate the activation process. The transmitter is initially silent. The activation request signal (defined in the HDSL specifications) is always considered to be true (i.e., ACTREQ = 1). The activation state is then changed to ACTIVATING_STATE.

Bit Pump Code—The bit pump ASM is idle during INACTIVE_STATE; the bit pump waits for the _ACTIVATE API command. The _ACTIVATE API command initializes the bit pump for the activation process.

3.9.3 ACTIVATING_STATE

ASM—In ACTIVATING_STATE, the ASM monitors the bit pump activation status. If the ACTivation Timer (T-Act) expires, the activation state is changed to DEACTIVATED_STATE. If the Noise Margin Reading OK (nmr_ok = 1) and Transmit 4-Level Indicator (Tx 4-Level = 1) flags are detected, the activation state is first changed to ACTIVATING_STATE_S1 and then to ACTIVE_RX_STATE.

Bit Pump Code—The pump code starts transmitting the 2-level (S0) signal and monitors for a received signal. When a signal is detected, the code clears the Loss Of Signal (LOS) flag and performs an optimize phase search, adapts filters, etc. The bit pump code then transmits the 4-level (S1) signal and monitors the received signal for S1. When the S1 signal is detected, the bit pump code performs final adaptation, at which time the nmr_ok and Tx 4-Level flags become valid.

3.9.4 ACTIVE_RX_STATE

ASM—In ACTIVE_RX_STATE, the ASM monitors the bit pump activation status. If the T-Act timer expires, the activation state is changed to DEACTIVATED_STATE. If the normal operation flag is detected, the activation state is changed to GOTO_ACTIVE_TX_RX_STATE.

Bit Pump Code—The bit pump ASM finalizes its activation process and sets the normal operation flag when it completes startup.

3.9.5 GOTO_ACTIVE_TX_RX_STATE

ASM—GOTO_ACTIVE_TX_RX_STATE serves as a common exit point for two states in the application ASM with a channel unit. It is not required for a bit pump-only application. In this state, the _TRANSMIT_EXT_DATA API command is called, which switches the transmitted data path from internal to the bit pump to external.

Bit Pump Code—The bit pump software processes the _TRANSMIT_EXT_DATA API command.

3.9.6 ACTIVE_TX_RX_STATE

ASM—In ACTIVE_TX_RX_STATE, the ASM monitors the bit pump activation status. If the los = 1 or nmr_ok = 0 (noise margin < –5 dB) flag is detected, the activation state is changed to PENDING_DEACTIVATED_STATE.

The nmr_ok bit is polled by the software. Whenever the ASM executes, it calls the _STARTUP_STATUS API command. This API command calls the function *_ReadNmr()* that reads the Noise Level Meter (NLM) register and performs a table lookup to convert the NLM reading into an NMR. The API command checks the NMR value and sets or clears the nmr_ok bit.

An LOS condition causes a bit pump interrupt. The interrupt handler copies the irq_source register to the *_int_reg[]* global variable, then the *_HandleFlags()* function polls *_int_reg[]* to check for changes such as LOS.

NOTE: It is recommended that a high layer protocol (e.g., HDLC idle frames) be added to detect a bad link condition. For certain loop conditions, the bit pump cannot guarantee 100% reliability in detecting a bad link. The HDLC check is added as the third conditional to the other two already in the software:

```
bp_status.bits.los || !bp_status.bits.nmr_ok || !hdlc_status_ok
```

Bit Pump Code—*_HandleTempEnv()* processes any temperature and environmental changes. Different status responses are continuously monitored and can be probed by issuing the corresponding status request commands.

3.9.7 PENDING_DEACTIVATED_STATE

ASM—In the pending deactivated state, the ASM monitors the bit pump activation status. If both los = 0 (return of signal) and nmr_ok = 1 (noise margin > –5 dB) flags become valid again within 2 seconds, the activation state is changed back to ACTIVE_TX_RX_STATE. If the los = 1 or nmr_ok = 0 flag is still set after 2 seconds, the activation state is changed to DEACTIVATED_STATE.

Bit Pump Code—*_HandleTempEnv()* processes any temperature and environmental changes. Different status responses are continuously monitored and can be probed by issuing the corresponding status request commands.

3.9.8 DEACTIVATED_STATE

ASM—In DEACTIVATED_STATE, the ASM issues the _DEACTIVATE API command that turns off the bit pump's transmitter. The ASM then waits for LOST = 1. When LOST = 1 becomes true, the activation state is changed to WAIT_FOR_LOST.

Bit Pump Code—The *_HandleFlags()* function starts the LOS Timer (LOST) when the _DEACTIVATE API command is received. This timer uses start-up timer 3 (sut3) within the bit pump, which provides high accuracy.

3.9.9 WAIT_FOR_LOST

ASM—In the WAIT_FOR_LOST state, the ASM waits for a 4-second time-out, which gives the HTU-R time to detect an LOS condition and turn off its transmitter. This delay allows the ASMs on the HTU-C and HTU-R to stay in sync.

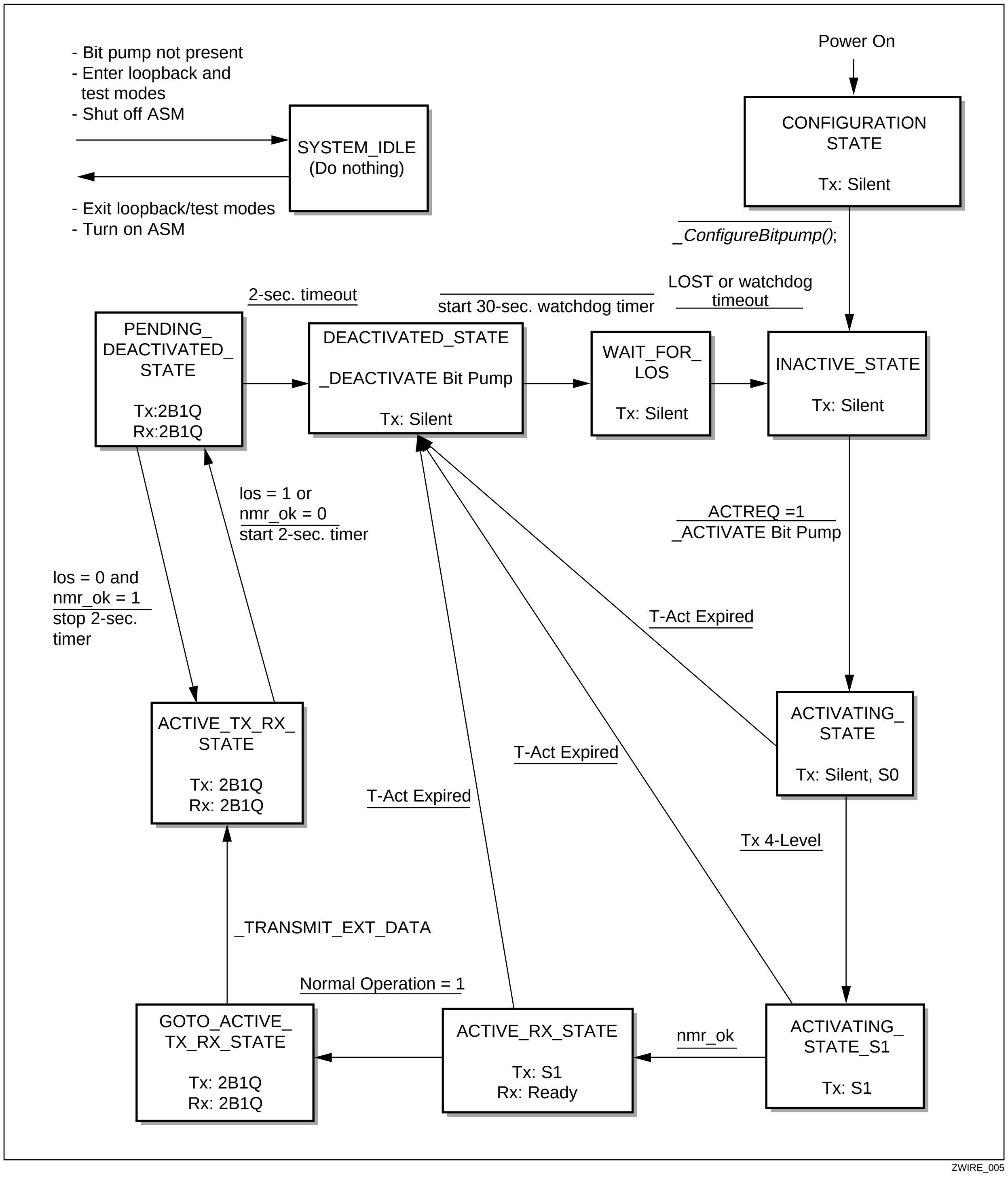
If there is a significant amount of noise on the DSL, the LOST may never occur. If the 30-second watchdog timer expires, the ASM proceeds to INACTIVE. This does not guarantee that the subsequent startup will succeed, but keeps the ASM out of an endless loop.

Bit Pump Code—The *_HandleFlags()* function sets the LOST flag (LOST = 1) when the LOST expires.

3.10 HTU-R Activation

Figure 3-4 illustrates the state diagram for the activation of the HTU-R.

Figure 3-4. HTU-R Activation State Machine



3.10.1 CONFIGURATION_STATE

ASM—In CONFIGURATION_STATE, the ASM calls the function *_ConfigureBitpump()* discussed in Section 3.3.

Bit Pump Code—The bit pump code processes the API commands.

3.10.2 INACTIVE_STATE

ASM—In INACTIVE_STATE, the _ACTIVATE API command is issued to initiate the activation process. The transmitter is initially silent. The activation state is then changed to ACTIVATING_STATE.

Bit Pump Code—The bit pump ASM is idle during INACTIVE_STATE; the bit pump waits for the _ACTIVATE API command. The _ACTIVATE API command initializes the bit pump for the activation process. The bit pump code then waits until an S0 signal is detected (los = 0).

3.10.3 ACTIVATING_STATE

ASM—In ACTIVATING_STATE, the ASM monitors the bit pump activation status. If the T-Act timer expires, the activation state is changed to DEACTIVATED_STATE. If the Noise Margin Reading OK (nmr_ok = 1) and Transmit 4-Level Indicator (Tx 4-Level = 1) flags are detected, the activation state is changed to ACTIVATING_STATE_S1 and then to ACTIVE_RX_STATE.

Bit Pump Code—The bit pump ASM waits until an S0 signal is detected (los = 0). When the signal is detected, the bit pump performs the frequency lock, searches for the optimum phase, adapts filters, etc. The bit pump code then begins transmitting the S0 signal and monitors, the received signal for S1. When the S1 signal is detected, the bit pump code performs some adaptations and starts transmitting the S1 signal. At this point, the nmr_ok and Tx 4-Level flags become valid.

3.10.4 ACTIVE_RX_STATE

ASM—In ACTIVE_RX_STATE, the ASM monitors the bit pump activation status. If the T-Act timer expires, the activation state is changed to DEACTIVATED_STATE. If the normal operation flag is detected, the activation state is changed to GOTO_ACTIVE_TX_RX_STATE.

Bit Pump Code—The bit pump ASM finalizes its activation process and sets the normal operation flag when it completes startup.

3.10.5 GOTO_ACTIVE_TX_RX_STATE

ASM—GOTO_ACTIVE_TX_RX_STATE serves as a common exit point for two states in the application ASM that includes a channel unit. It is not required for a bit pump-only application. In this state, the _TRANSMIT_EXT_DATA API command is called, which switches the transmitted data path from internal to the bit pump to external.

Bit Pump Code—The bit pump software processes the _TRANSMIT_EXT_DATA API command.

3.10.6 ACTIVE_TX_RX_STATE

ASM—The ASM monitors the bit pump activation status. If the los = 1 or nmr_ok = 0 (noise margin reading < –5 dB) flag is detected, the activation state is then changed to PENDING_DEACTIVATED_STATE.

The nmr_ok bit is polled by the software. Whenever the ASM executes, it calls the _STARTUP_STATUS API command. This API command calls the function *_ReadNmr()* that reads the Noise Level Meter (NLM) register and performs a table lookup to convert the NLM reading into a NMR. The API command checks the NMR value and sets or clears the nmr_ok bit.

An LOS condition causes a bit pump interrupt. The interrupt handler copies the irq_source register to the *_int_reg[]* global variable, then the *_HandleFlags()* function polls *_int_reg[]* to check for changes such as LOS.

NOTE: It is recommended that a high layer protocol (e.g., HDLC idle frames) be added to detect a bad link condition. For certain loop conditions, the bit pump cannot guarantee 100% reliability in detecting a bad link. The HDLC check is added as the third conditional to the other two already in the software:

```
bp_status.bits.los || !bp_status.bits.nmr_ok || !hdlc_status_ok
```

Bit Pump Code—*_HandleTempEnv()* processes temperature and environmental changes to the DSL. Different status responses are continuously monitored and can be probed by issuing the corresponding status request commands.

3.10.7 PENDING_DEACTIVATED_STATE

ASM—The ASM monitors the bit pump activation status. If both the los = 0 (return of signal) and nmr_ok = 1 (noise margin > –5 dB) flags become valid again within 2 seconds, the activation state is changed back to ACTIVE_TX_RX_STATE. If the los = 1 or nmr_ok = 0 flag is still set after 2 seconds, the activation state is changed to the DEACTIVATED_STATE.

Bit Pump Code—*_HandleTempEnv()* processes temperature and environmental changes to the DSL. Different status responses are continuously monitored and can be probed by issuing the corresponding status request commands.

3.10.8 DEACTIVATED_STATE

ASM—In DEACTIVATED_STATE, the ASM issues the _DEACTIVATE API command which turns off the bit pump's transmitter. The activation state is changed to WAIT_FOR_LOS.

Bit Pump Code—The bit pump software does not perform any processing.

3.10.9 WAIT_FOR_LOS

ASM—In WAIT_FOR_LOS state, the ASM waits for an LOS condition. This ensures that the HTU-C turns off its transmitter before the HTU-R starts the training procedure.

If there is a significant amount of noise on the DSL, the LOS condition may never occur. If the 30-second watchdog timer expires, the ASM proceeds to INACTIVE. This does not guarantee that the subsequent startup will succeed, but keeps the ASM out of an endless loop.

Bit Pump Code—The *_HandleFlags()* function sets the los flag when the bit pump interrupt signals a low Far-End Level Meter (FELM) condition.

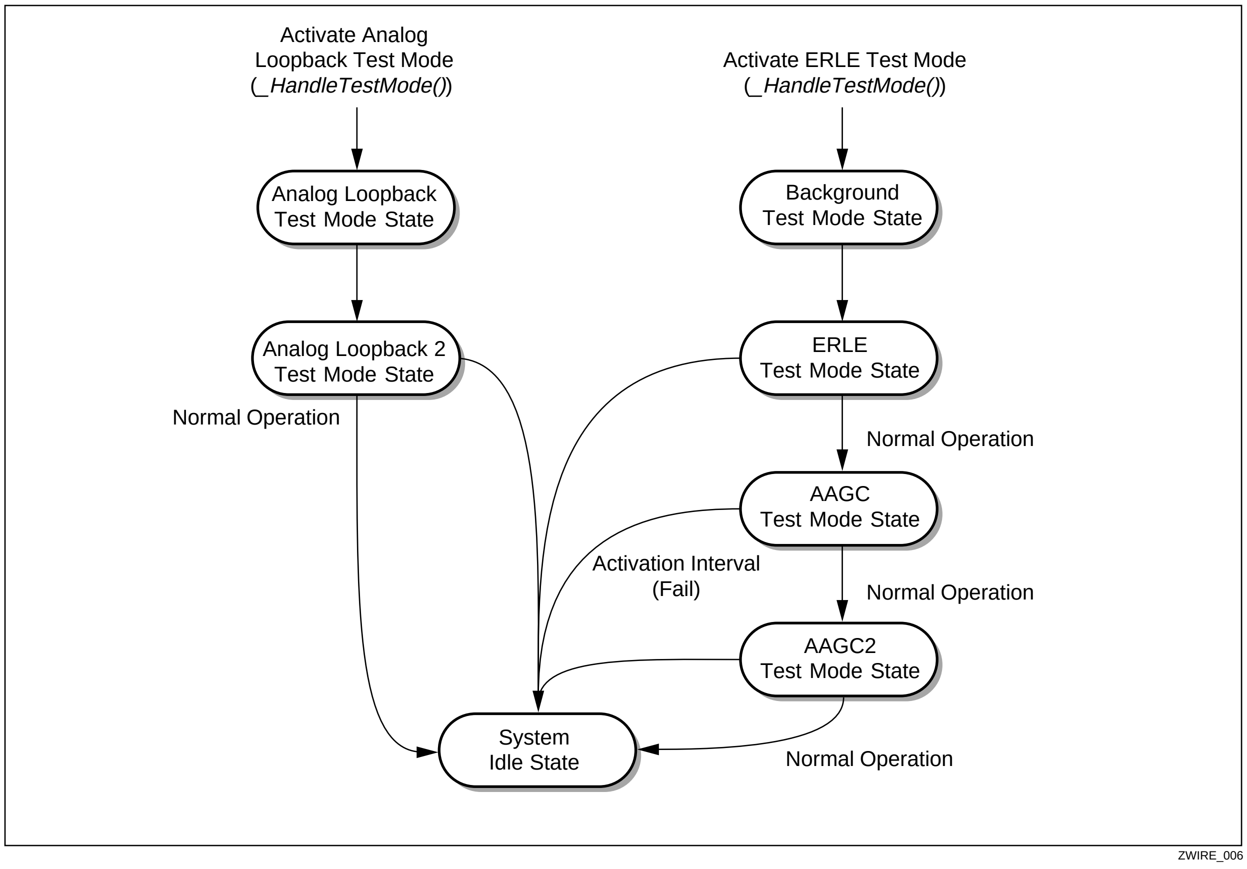
3.11 ERLE and Analog Loopback Test Modes

As illustrated in Figure 3-5, the application ASM contains additional states to run the ERLE and analog loopback tests. These test modes are integrated into the ASM to minimize the real-time constraints of these features. The *_HandleTestMode()* state machine executes in parallel to the application ASM. The *_HandleTestMode()* function updates the normal_operation and activation_interval status flags to the ASM as it proceeds through the test modes. Figure 3-5 illustrates that these status flags are checked by the ASM.

In the analog loopback tests, the terminal performs an activation procedure using its own transmitted signal as the received signal. The activation process is required because the DSP filters are part of the loopback signal path.

The ERLE test mode is discussed in Section 8.5.

Figure 3-5. Application ASM States for ERLE and Analog Loopback Test Modes


Activate Analog Loopback Test Mode (_HandleTestMode())
Activate ERLE Test Mode (_HandleTestMode())
Analog Loopback Test Mode State
Background Test Mode State
Analog Loopback 2 Test Mode State
ERLE Test Mode State
Normal Operation
Normal Operation
AAGC Test Mode State
Activation Interval (Fail)
Normal Operation
AAGC2 Test Mode State
System Idle State
Normal Operation
ZWIRE_006

3.12 _HandleReceiveMessages() and _HandleTransmitMessages()

In a dual processor architecture, the ZipWire software handles one end of the serial communication protocol. As illustrated in Figure 2-1, this interface includes a serial interrupt function, which receives or transmits 1 byte over the serial link, and the serial communication functions *_HandleReceiveMessages()* and *_HandleTransmitMessages()* that handle received and transmitted messages, respectively.

The *_HandleReceiveMessages()* function waits until a complete 4-byte message (with legal header and checksum bytes) is received, then executes the command by calling the appropriate API function. The header and checksum bytes ensure successful resynchronization of the serial message transfer protocol if the 4-byte message boundary goes out of sync. It also reduces the probability of false command interpretation.

The *_HandleTransmitMessages()* function controls the serial transmission of messages waiting in a transmit messages buffer.

4.0 Bit Pump Software Operation

This chapter outlines the operation of the bit pump-specific software. Emphasis is given to subjects where some knowledge of the code operation may be beneficial when using, integrating, or writing code for the bit pump.

4.1 _BtMain() Function

_BtMain() is the main function that executes all bit pump-related tasks. A copy of the code, minus some comments, is as follows:

```
void _BtMain (BP_U_8BIT no)
{
   status_reg_type status_reg;                         /* Status variable */
   user_setup_low_type user_setup_low;
   user_setup_high_type user_setup_high;

   DECLARE_PTR;

   if (_bp_vars[no].bp_flags.bits.operational) /* Bit_pump #no operational */
     {
     INIT_BP_PTR;

     _HandleFlags(no);
     RD_BYTE(no, STATUS, status_reg.status);
     if ( !status_reg.bits.normal_operation )
       {
       RD_WORD(no, USER_SETUP, user_setup_low.setup, user_setup_high.setup);
       if ( user_setup_low.bits.terminal_flag )                   /* HTU-R */
         _HturControlProcess(no);                /* HTU-R control process */
       else                                                       /* HTU-C */
         _HtucControlProcess(no);                /* HTU-C control_process */

       _HandleTestMode(no);
       }                                               /* end ! normal op */
   else
       {
#ifdef TEMP_ENV
   _HandleTempEnv(no);
#endif
#ifdef ZIP_START
   _ZipStart_HandleUpdate(no);
#endif
       }
   }                                   /* END-IF bit_pump #no is operational */
}                                                       /* END _BtMain() */
```

The application's main program calls the *_BtMain()* function, which is responsible for executing all the bit pump tasks, including the bit pump ASM. *_BtMain()* is the highest-level block in the bit pump software hierarchy and must be executed periodically within specified real-time constraints (see Chapter 6.0 for details).

Because *_BtMain()* is passed a bit pump number, it services one bit pump at a time. In earlier versions of the ZipWire software, *_BtMain()* contained a for loop to service all the bit pumps. However, maintaining responsiveness in an application with many bit pumps (e.g., 28 bit pumps) requires the flexibility to control the execution rate of *_BtMain()* for a given bit pump. One possibility is to set the execution rate based on the status (startup, normal operation, out-of-service) of a bit pump.

Before calling any functions, *_BtMain()* checks the operational flag. This flag is set during initialization by the Bit Pump On/Off API command (see Section 3.2). Based on the normal_operation flag, *_BtMain()* calls one of the bit pump ASM functions or calls *_HandleTempEnv()*. This flag is set by the bit pump ASM functions when they complete startup and reach the SET_NORMAL_CONDITIONS state.

The *_HtucControlProcess()* and *_HturControlProcess()* functions implement the bit pump ASMs (for the HTU-C and HTU-R) and must be executed periodically during startup. These functions go through the complete activation procedure.

The *_HandleTestMode()* function implements the test mode state machine (see Figure 3-5) which runs the ERLE test mode (see Section 8.5) and analog loopback test modes (see Section 8.3.2).

The *_HandleTempEnv()* function handles any temperature and environmental changes to the DSL. The *_HandleTempEnv()* function runs continuously after the bit pump reaches normal operation (see Section 6.8).

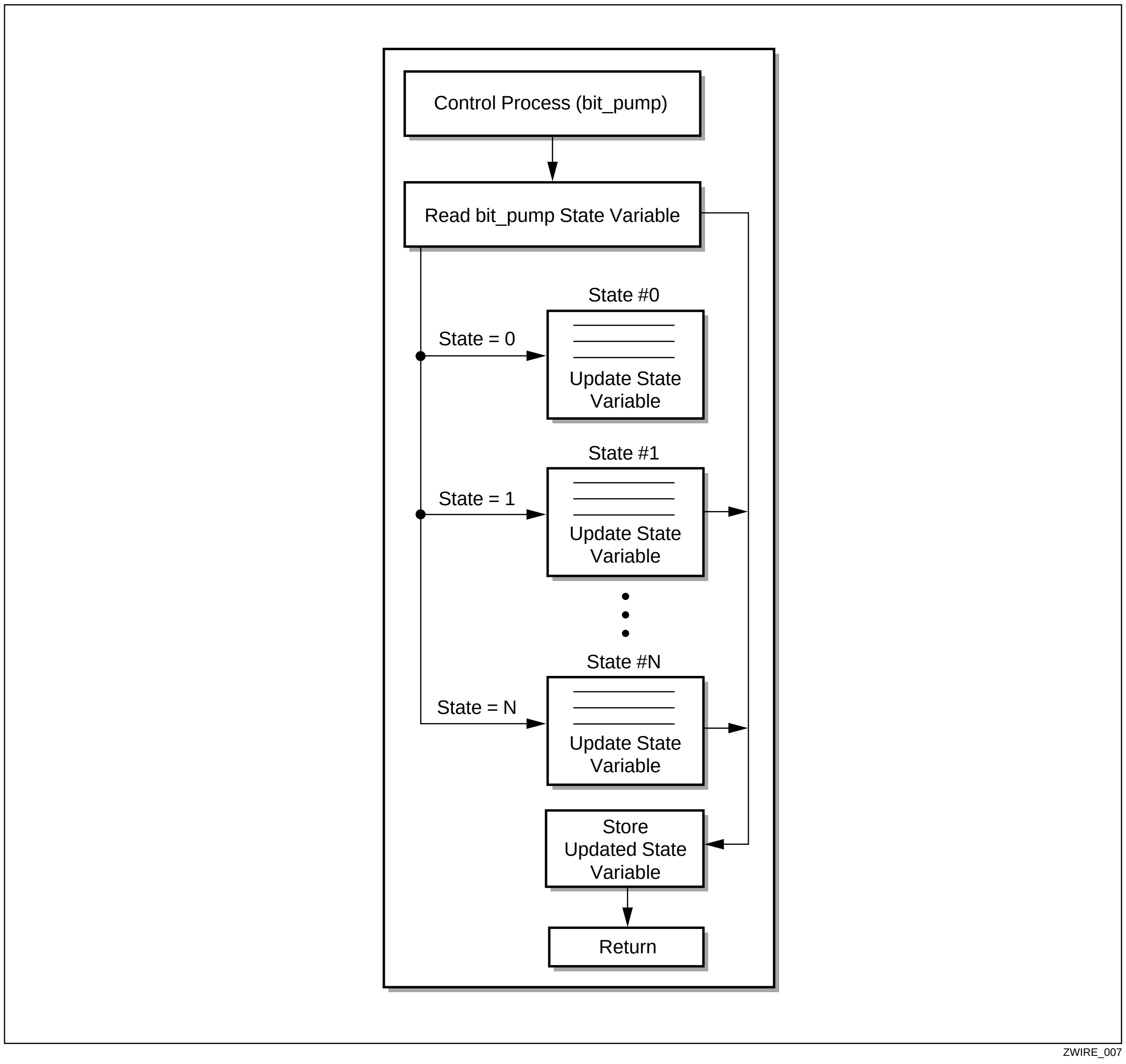
The DECLARE_PTR and INIT_BP_PTR macros in Figure 4-1 are discussed in Section 5.9.4. The RD_WORD macro is discussed in Section 4.2.

4.2 Bit Pump Activation State Machines

The bit pump ASM functions (*_HtucControlProcess()* and *_HturControlProcess()*) are the largest and most important functions within the ZipWire software. Each function contains the code for activating one bit pump of a specified terminal type, where the bit pump number is passed as an input parameter.

_HtucControlProcess() and *_HturControlProcess()* functions contain the activation and normal operation states. Figure 4-1 illustrates a high-level flowchart for the *_HtucControlProcess()* and *_HturControlProcess()* functions.

Figure 4-1. Operation of the Bit Pump ASM Functions, _HturControlProcess(), and _HtucControlProcess()



The current state of each active bit pump is stored in a global STAGE variable. When the ASM begins execution, this variable is read using the RD_WORD macro:

```
RD_WORD(no, STAGE, stage, tm_stage);
```

Every execution of the bit pump ASM (with a specific bit pump index) results in the execution of the control operations that correspond to the current bit pump state. The processing within a state may include several basic control operations. The processing for only one state for one bit pump is executed each time the ASM is executed. The same function is used for all active bit pumps, only the state variable being bit pump-specific. After completing the execution of all operations belonging to the present state, the local stage variable is stored in the global STAGE variable using the WR_WORD macro:

```
WR_WORD(no, STAGE, stage, tm_stage);
```

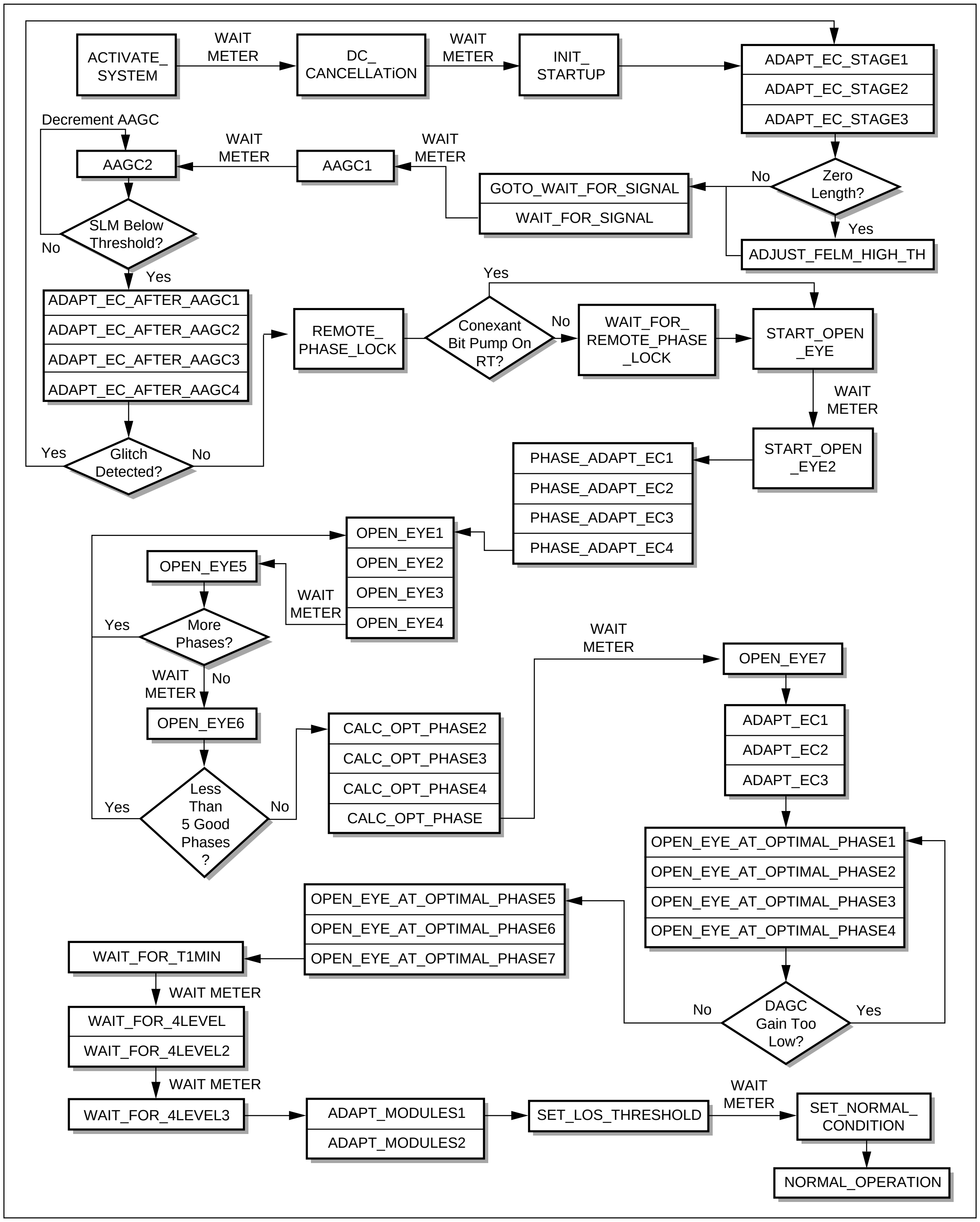
The succeeding state operations are executed the next time the ASM function is executed with the same bit pump index parameter. Once activation is completed, the bit pump goes to the NORMAL_OPERATION state, where it stays until a reset or reactivation occurs.

The activation state is stored in the upper byte of the global 2-byte STAGE variable. The lower byte contains the state of the separate test-mode state machine. Many states transition to the WAIT_FOR_METER state as an intermediate polling state. The next state after WAIT_FOR_METER is stored in the lower byte of the global 2-byte STAGE2 variable.

Figure 4-2 and Figure 4-3 illustrate detailed state diagrams for the HTU-C and HTU-R, respectively. These state diagrams are presented as background information because the customer is not responsible for maintaining the bit pump software. However, this background information is useful when a customer wants a deeper understanding or reports a problem and Conexant requests debugging information such as a TDEBUG output (see Section 8.2).

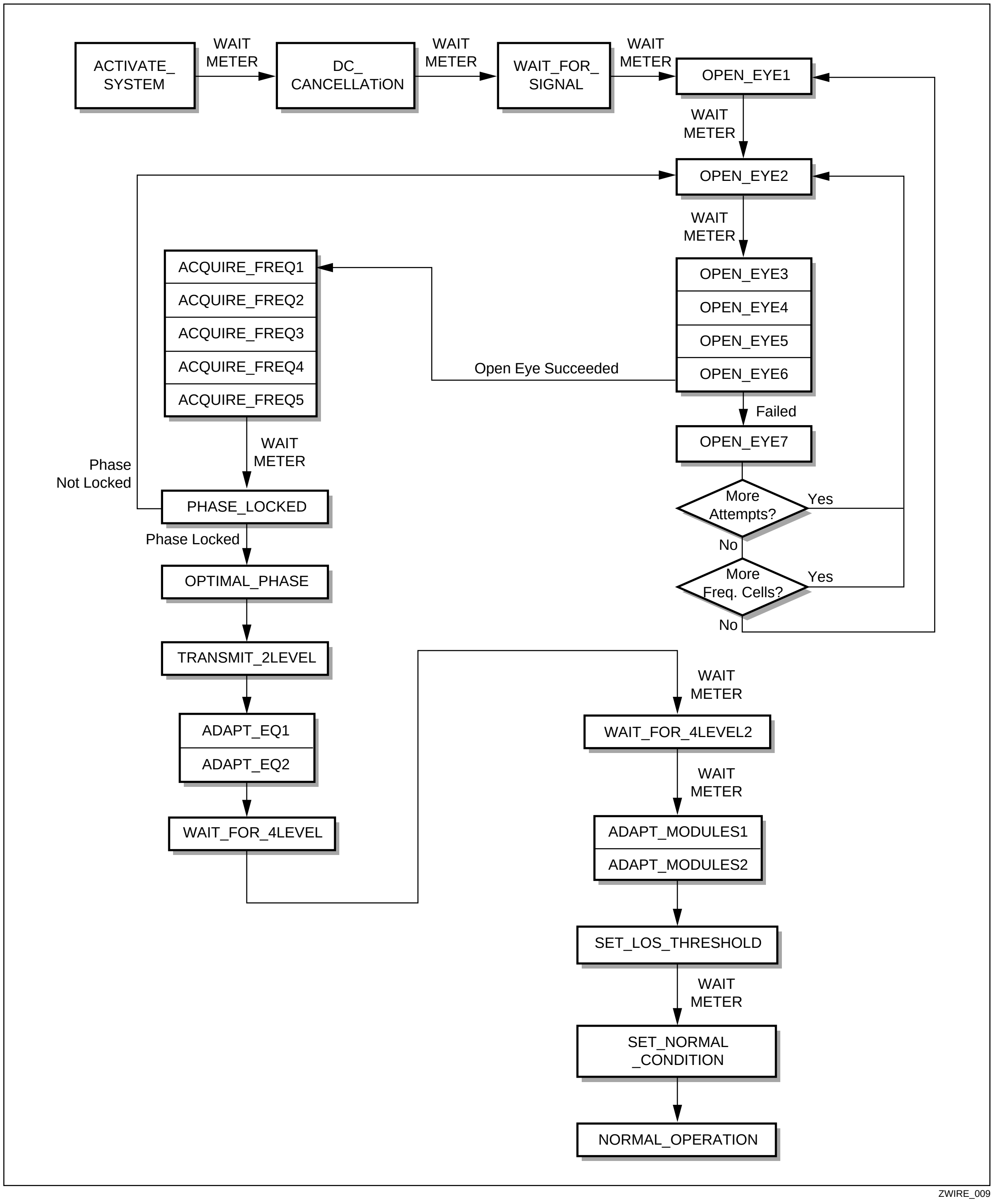
The bit pump ASM does not implement any watchdog timers; rather, it uses a “best try” philosophy. As illustrated in Figures 4-2 and 4-3, the bit pump ASM attempts to correct for errors by reverting back to a previous state. In these scenarios, the software typically loads a different set of coefficients. The software continues to attempt the activation until the activation timer in the application ASM expires.

Figure 4-2. Detailed Bit Pump ASM (HTU-C)



ZWIRE_008

Figure 4-3. Detailed Bit Pump ASM (HTU-R)



4.3 Bit Pump Interrupt Handling

_BpInterruptHandler() responds to interrupts initiated by any of the bit pumps. Because the interrupt signals from different bit pumps are hardwire-ORed, the interrupt handling function identifies the source of the interrupt by reading the bit pump interrupt status registers of all active bit pumps. For most interrupt sources, the interrupt handling function only sets an appropriate software interrupt status flag. Certain emergency interrupts receive treatment in the interrupt handling function itself. All other interrupts are handled by the bit pump ASM function or by the *_HandleFlags()* function which is called by *_BtMain()*.

If a customer has unique requirements (such as many bit pumps), the software can be modified to optimize interrupt processing. One possibility is to poll the bit pumps for their statuses; another is to modify the bit pump interrupt to mask the meter interrupt during startup. The meter interrupt is a continuous type of interrupt which causes an endless stream of interrupts once it is unmasked. The meter interrupt is the only continuous type of interrupt; the timer interrupts are “one-shot” type interrupts. Contact Conexant for additional information on ways to optimize the bit pump software for multiple bit pump applications.

During startup, the bit pump interrupts are initialized in the following sequence:

1. *_MaskBtHomerInt()*—Masks all interrupts.
2. *_Init8051()*—Enables the microcontroller interrupt.
3. *_EnableBitpump()*—Calls *_BtControl(bp,_SYSTEM_CONFIG,_PRESENT).*
4. *_BtControl(bp,_SYSTEM_CONFIG,_PRESENT)*—Sets *_bp_vars[no].bp_flags.bits.operational* and then calls *_BtInitialize()*.
5. *_BtInitialize()*—Calls *ClearBitpumpIntSource().*
6. *ClearBitpumpIntSource()*—Unmasks all interrupts in mask_low_reg(0x02) except the meter timer.
7. *_HturControlProcess()* and *_HtucControlProcess()*—Calls *_SetMeterTimer()*.
8. *_SetMeterTimer()*—Unmasks the meter timer. The bit pump interrupts begin.

The rate of the bit pump interrupt depends on the meter timer interval and the data rate. The rate is set based on the parameter passed in the *_SetMeterTimer()* function. Table 4-1 lists the rates for the parameters based on a 784 Kbps data rate.

The interrupt rates (or any value based on symbols) can be computed using the following method. First, compute the symbol time based on the data rate:

$$1\text{ symbol time (ms)} = (2\text{ bits} \mathbin{/} \text{symbol}) \times (1 \mathbin{/} \text{data rate in Kbps})$$

For example, the symbol time at a data rate of 1,168 Kbps:

$$1\text{ symbol time (ms)} = (2\text{ bits} \mathbin{/} \text{symbol}) \times (1 \mathbin{/} 1{,}168\text{ Kbps}) = 1.71^{-03}\text{ms}$$

Then, the total time is computed based on the number of symbols. For 32,768 symbols, use the following computation:

$$\text{time for } 32{,}768 \text{ symbols (ms)} = 32{,}768 \times 1 \text{ symbol time}$$

The total time for 32,768 symbols at a data rate of 1,168 Kbps is as follows:

$$\text{time for } 32{,}768 \text{ symbols (ms)} = 32{,}768 \times 1.71^{-3}\text{ms} = 56\text{ ms}$$

Customers can debug their systems by checking the interrupt rates without a DSL connection. At startup, the bit pump ASMs remain at the WAIT_FOR_SIGNAL state without a DSL connection. The interrupt rates on the HTU-R and HTU-C are set to the ALT_METER value in the WAIT_FOR_SIGNAL state.

Table 4-1. Bit Pump Interrupt Rates for _SetMeterTimer Parameters

_SetMeterTimer() Parameter	# Symbols	Interrupt Rate (ms) at 784 Kbps data rate
DEFAULT_METER	1,024	2.125
PHASE_LOCK_METER	1,024	2.125
FOUR_LEVEL_METER	2,048	5.25
PHASE_QUALITY_METER	4,096	10.5
ALT_METER	8,192	21
NORMAL_METER	32,768	84

4.4 Bit Pump Timers

The bit pump device has many timers that are available for the microcontroller. Table 4-2 lists these timers and their uses by the software, if any.

Table 4-2. Bit Pump Device Timers

Bit Pump Timer	Before Startup	During Startup	After Startup	During Test Modes
t4[(1)]	Unused	Unused	Unused	Unused
t3	Unused	Adapt EC, Filters, etc.	Adapt filters	Adapt filters
snr	Unused	Unused	Unused	Unused
meter	Meters	Meters	Meters	Meters
sut4	Unused	Activation Timer	Unused	Time-out
sut3	Unused	T1 Min[(2)]	LOST	Unused
sut2	Unused	T4 Max[(2)] and Adaptions	Unused	Unused
sut1	Unused	Unused	Unused	Adaptions during ERLE

NOTE(S):
(1) Bit pump timer T4 has no relationship to the timer T4 Max defined in the HDSL specifications.
(2) See the HDSL specification.

When a bit pump timer expires, the bit pump interrupts the microcontroller. *_BpInterruptHandler()* copies the bit pump timer_source register (0x04) to the timer_source global variable in *_int_reg[]*. Another function or macro (as listed in Table 4-3) then polls this global variable (using the **int_ptr* pointer) to check for status changes.

Table 4-3. Functions and Macros That Poll Bit Pump Timers

Bit Pump Timer	Polling Function or Macro
t3	TIMER_BREAK(t3)
meter	TIMER_BREAK(meter)
sut4	*_HandleFlags()*
sut3	*_HandleFlags()*
sut2	TIMER_BREAK(sut2)
sut1	TIMER_BREAK(sut1)

4.5 Application-Specific Software Timers

A general purpose timer is implemented using the 80C32 timer 0 (timer 1 is used as a baud rate generator). The timer structure is implemented as a two-dimensional array to index multiple bit pumps and general purpose timers. Currently, two general purpose timers per bit pump are used; one is a pending deactivation timer and the other is a 1-second timer. To add more timers, increase the _NO_GEN_PURPOSE_TIMERS definition and add the appropriate timer index definitions. The timer data structures are located in timer.h. The timer variables are declared in timer.c.

The data structure for each general purpose timer includes a status byte that is defined in Table 4-4. The *_InitGenPurposeTimer()* function initializes the 80C32 timer 0 and sets the status byte to 0.

Table 4-4. General Purpose Status Bit Definitions

Bit	Description	0 Value	1 Value
0	State	Disabled	Enabled
1	Complete	Not expired	Expired
2	Continuous	Not continuous	Continuous
3–7	Reserved	—	—

4.5.1 One-Second Timer

The 1-second timer is a continuous mode timer. In this mode, the timer continuously counts the specified time interval. The typedef data structure GEN_PURPOSE_CONT_TIMER is defined for any continuous mode timer:

```
typedef struct
{
    BP_U_16BIT load_value;
    BP_U_32BIT elapsed_counter;
} GEN_PURPOSE_CONT_TIMER;
```

The *elapsed_counter* variable stores the number of times the continuous timer has expired. The *load_value* variable sets the timer interval when the continuous timer is reloaded. The *_ContinuousGenPurposeTimer()* function enables or disables a continuous mode timer.

The 1-second timer is used to:

- estimate the bit pump startup time
- run the watchdog timer in the WAIT_FOR_LOS and WAIT_FOR_LOST states
- determine the time to save the ZipStartup coefficients

4.5.2 Pending Deactivation Timer

The pending deactivation timer is a "one-shot" timer set to 2 seconds; it is not a continuous mode timer. When this timer expires, the application ASM transitions from PENDING_DEACTIVATED_STATE to DEACTIVATED_STATE.

On the EVM, the pending deactivation timer also provides a 1-second delay to allow time to press the reset button on the other terminal. The 1-second delay allows the application ASMs on the HTU-R and HTU-C to start up in sync.

5.0 Compiling the Bit Pump Source Code

This chapter contains detailed information on how to modify and compile the bit pump source code to match the specific application environment. The application environment includes the:

- microprocessor or microcontroller
- compiler
- hardware

In addition to source code modifications, compilation directives must be correctly defined at compile time. To create fully working executable code, follow these steps:

1. Determine the compiler directives required for the environment (see Section 5.7).
2. Modify or write all the modifiable sections by following the instructions in Sections 5.8 and 5.9.
3. Compile and link the source files. Use the file list and dependency information in Sections 5.3–5.6.

5.1 Keil Tool Set

The EVM code is compiled with the Keil 8051 C Compiler V5.20 Toolset. The uVision Integrated Development Environment (IDE) includes project files which specify the build list. The uVision IDE does not use a separate makefile because it automatically checks for file dependencies.

The SCRIPT.BLD (Intersolv Build Utility) is also provided to maintain compatibility with previous releases. The SCRIPT.BLD file can be used as a reference for customers who must create a makefile.

5.2 RAM and ROM Requirements

Table 5-1 lists the RAM and ROM requirements for two build options. The code was compiled with the Keil C compiler V5.20 using the small memory model and these Keil-specific compiler switches: SB, DB, OE, and NOAM. Table 5-1 lists ZipWire-specific compiler directives.

The RAM requirements do not include the stack size; an additional 24 bytes is required for the run-time stack. These requirements include the Activation State Diagram application example code.

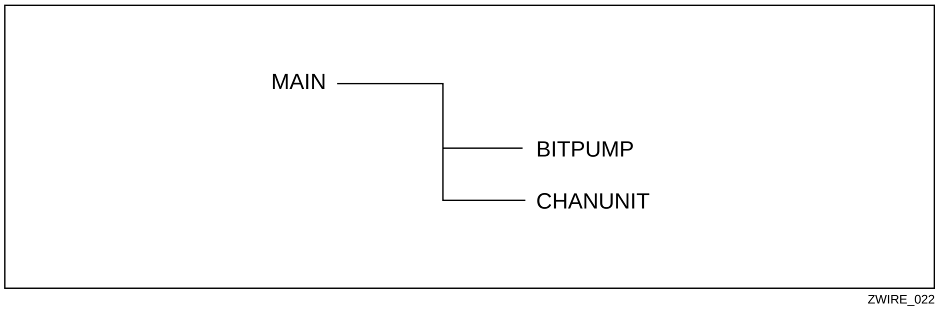
Table 5-1. ROM and RAM Requirements for Various Build Options

Build Option	Compiler Directives	RAM (Bytes)	XDATA RAM (Bytes)	ROM (Bytes)
Minimum HTU-C	C51,PDATA_MODE,ADD_DELAY,HTUC	116	85	27,965
Minimum HTU-R	C51,PDATA_MODE,ADD_DELAY,HTUR	117	85	27,418

5.3 Directory Structure

As illustrated in Figure 5-1, the directory structure is partitioned into three directories: the MAIN (application) directory as the root, the bit pump directory as a subdirectory labeled BITPUMP, and the channel unit directory as a subdirectory labeled CHANUNIT.

Figure 5-1. Directory Structure



The application software, hex files, Keil project files, etc., are located in the MAIN directory. All bit pump source code and header files are in the BITPUMP subdirectory. All channel unit source code and header files are in the CHANUNIT subdirectory.

NOTE: A separate subdirectory for the bit pump code is recommended to more easily manage the software and handle future upgrades. However, the final implementation of the directory and file structure is up to the user.

5.4 MAIN (Application) Directory

Table 5-2 lists all C source files found under the MAIN (application) directory.

Table 5-2. Source Files Under the Main Directory

Files	Description
dsl_main.c	Main Program for HDSL EVM System
dsl_init.c	HDSL Initialization Functions
dsl_asm.c	HDSL Activation State Machine
dsl_man.c	HDSL Dynamic Loop Control Functions (requires defining the CHAN_UNIT directive)
dsl_misc.c	HDSL Miscellaneous Functions
dsl_api.c	HDSL Application Level API Commands (requires defining the CHAN_UNIT directive)
bt_api.c	HDSL API Commands Parser
zipvalid.c	ZIPSTARTUP Validation Function (requires ZipStartup software package; see Section 2.8)
timer.c	80C32 General Purpose Timer Functions

5.5 BITPUMP Directory

Table 5-3 lists all C source files found under the BITPUMP subdirectory.

Table 5-3. Source Files Under BITPUMP Subdirectory

Files	Description
api.c	Bit Pump API Processing
btmain.c	Bit Pump Main Function
btint.c	Bit Pump Interrupt Handler
bitpump.c	Handle Any Bit Pump Interrupt Status and Temperature Changes
init51.c	Functions for Initializing 8051 Interrupts, Timers, and Serial Port
mail.c	Handle Receive Messages, Transmit Messages, Manage Read/Write to Mail Boxes
monitor.c	RS232 Printf Support (requires defining the TDEGUG directive)
serint.c	Serial Communication Interrupt Handler
suc.c	Bit Pump ASM for the HTU-C
sur.c	Bit Pump ASM for the HTU-R
suutil.c	Startup Utility Functions That Are Common to HTU-C and HTU-R
testmode.c	Performs Test Modes: Loopbacks, Isolated Transmit Pulse, Scrambled 1s, VCXO Control Voltage Test; Performs Bit Pump Full Self-Test
user.c	User Modifiable Code. Definitions of the Absolute Addresses of Bit Pump Chips, and Function for Initializing the Bit Pump Pointers to These Addresses
util.c	Bit Pump Utility Functions
zipstart.c	Save and Load the ZipStartup Registers (requires ZipStartup software package; see Section 2.8)

5.6 Header File Dependencies

Table 5-4 lists header file dependencies for the application and bit pump software.

Table 5-4. Header File Dependencies

Files	Files Depend on…
Application Files	Application Header Files: dsl_main.h, dsl_init.h, dsl_incl.h, dsl_man.h, dsl_misc.h, dsl_api.h, bt_api.h, zipvalid.h, timer.h, typedefs.h Bit Pump Header Files: btmain.h, api.h, user.h, bitpump.h, suc.h, testmode.h, extern.h, zipstart.h, mail.h, init51.h Channel Unit Header Files: none, if CHAN_UNIT not declared
Bit Pump Files	Application Header Files: bt_api.h, typedefs.h Bit Pump Header Files: api.h, bitpump.h, btmain.h, extern.h, init51.h, mail.h, ptrdef.h, serint.h, suc.h, sur.h, suutil.h, testmode.h, user.h, util.h, zipstart.h Channel Unit Header Files: none
bthomer.h	btmain.h, mail.h, init51.h, intbug.h, serint.h, bitpump.h, suc.h, sur.h, suutil.h, util.h, api.h, testmode.h, user.h, ptrdef.h, typedefs.h, extern.h

5.7 Compiler Directives

The compiler directives select different code sections for compilation and are used in code sections that are either hardware-dependent or compiler-dependent. Accurate declaration of these directives is necessary to generate correctly working executable code.

Table 5-5 lists the directives defined in the Keil project files delivered with the ZipWire software. If SINGLE_LOOP and TWO_LOOPS are not defined, the default value of _NO_OF_LOOPS is set to 3 (see btmain.h in Table 5-4).

The CHAN_UNIT and REPEATER flags are discussed in the *RS8953B Application and Channel Unit Software Developer's Guide* (NCHUDG1B). These flags must not be defined for a bit pump-only application.

The ZipWire software defines these directives in bitpump.h: TEMP_ENV, BP_MSPACE, APPL_SW_MSPACE and not in any project file.

Table 5-5. Directives Defined in Keil Project Files

Keil Project File	**Directive**												
	C51	ADD_DELAY	PDATA_MODE	SER_COM	HTUR	HTUC	BER_METER	ERLE	SINGLE_LOOP	TWO_LOOPS	REPEATER	CHAN_UNIT	TDEBUG
ZIPWIRER	x	x	x	x	x	—	x	x	—	—	—	x	—
ZIPWIREC	x	x	x	x	—	x	x	x	—	—	—	x	—
ZIPREG	x	x	x	x	x	x	x	x	—	x	x	x	—
TDEBUG	x	x	x		x	x	x	x	—	—	—	—	x
NOTE(S): It is extremely important to recompile all source modules after defining these directives to ensure the changes propagate throughout the entire executable. Failure to do so will result in unpredictable behavior.													

AAGC15BUG (deleted in ZipWire Software Version 3.0)

The AAGC15BUG directive causes the software to not select the 15 dB setting as the final AAGC setting. In Version 3.0, the AAGC15BUG directive was deleted. Alternatively, Version 3.0 changed the SIGNAL_LEVEL_TH threshold from 6,500 to 5,300 (2 dB less than 6,500) to prevent A/D overflows. In addition, other algorithmic changes were made to meet the ETSI and ANSI requirements for 784 and 1,168 Kbps data rates. This alternative solution should allow a greater maximum reach on loops that have good analog echo cancellation and therefore can take advantage of the AAGC 15 dB setting.

ADD_DELAY

Defining the ADD_DELAY directive causes the addition of a 2-symbol delay before accessing specific addresses in the bit pump. This additional delay is required when a read/write operation to the bit pump is performed in less than two symbol periods. The delay is added only for very specific bit pump addresses where the read/write operations are limited to one access for each two-symbol period. The ADD_DELAY directive should be declared only when the external read/write cycle time of the microprocessor is faster than the symbol rate. When this directive is defined, the ZipWire software meets the timing requirements for the 80C32 processor running at 11 MHz and data rates down to 128 Kbps. When a faster processor is used, additional No OPs (NOPs) must be added to the *_Delay2Symbols()* function in user.c.

APPL_SW_MSPACE, BP_MSPACE (added in ZipWire Software Version 3.0)

The APPL_SW_MSPACE and BP_MSPACE directives in typedefs.h determine whether the application and bit pump global variables are located in internal or external RAM. These directives make it easy to move the variables from one memory space to another. The ZipWire EVM places these variables into external memory by defining these directives:

```
#define BP_MSPACE BP_XDATA
#define APPL_SW_MSPACE BP_XDATA
```

Placing the variables in external RAM prevents internal RAM constraints when the channel unit software is used. These directives and other directives that control memory options are discussed in Section 5.10.2. The BP_XDATA definition allows the specification of compiler-dependent memory spaces (see Table 5-7).

BER_METER

The BER_METER directive compiles in the bit pump BER meter code. The BER meter code adds ~400 bytes of ROM and 1 byte of RAM plus an additional 5 bytes of RAM for each bit pump. See Section 8.4 for more details.

NOTE: The BER_METER directive cannot be declared when the INT_BUG directive is specified.

BIT_REVERSE

The BIT_REVERSE directive controls the bit-field definitions in the bit pump software. Normally, when defining a C structure containing bit fields, such as:

```
typedef struct{
    unsigned char data_source:3;
    unsigned char htur_lfsr:1;
    unsigned char transmitter_off:1;
    unsigned char isolated_pulse:2;
    unsigned char:1;
}tx_mode_type;
```

the first bit is allocated to the least significant bit in a byte. This fact is critical when the bit fields are used to access external addresses, such as bit pump control bits.

The BIT_REVERSE directive should be defined if the compiler allocates the first bit field to the most significant bit in a byte.   This directive is usually required for Motorola-based systems.

Follow these steps to determine if the BIT_REVERSE directive is required:

1. Verify that the microcontroller can read the bit pump memory by reading the global_modes register (0x00) and confirming that its contents are valid.
2. Compile the source code without defining the BIT_REVERSE directive.
3. Set a breakpoint when writing a single bit to a bit pump register. For example, set a breakpoint on

   ```
   BP_WRITE_BIT(bp_mode_ptr, misc_test, async_mode, ASYNCHRONOUS_MODE);
   ```

   This line of code is in *_MaskBtHomerInt()* in btmain.c. This statement sets the least significant bit in register 0x0F.
4. Single step past the statement to make sure the statement has been executed.
5. Read the bit pump register that was modified. In this example, if register 0x0F = 0x01, BIT_REVERSE does not need to be defined. If register 0x0F = 0x80, BIT_REVERSE must be defined.

BP_MASK_INTERRUPTS (added in ZipWire Software Version 4.1)

The BP_MASK_INTERRUPTS directive supports systems that contain many bit pumps. This directive modifies the *_DisableBitpumpInterrupt()* function to mask the interrupt request from a single bit pump rather than masking the interrupt in the microcontroller for all bit pumps.

Masking the interrupt for all bit pump interrupts increases system startup time. This delay becomes large in multiple bit pump applications and may be unacceptable. This directive allows the interrupts of other bit pumps (that do not have their interrupts masked) to continue to have interrupts serviced.

This directive causes *_DisableBitpumpInterrupt()* to store the current interrupt masks in the *_bp_vars[]* structure. This storage requires 2 bytes per bit pump. *_DisableBitpumpInterrupt()* also sets a bit pump-dependent flag. This flag informs *_BtInterruptHandler()* to not poll the timer_source (0x04) or irq_source (0x05) registers for a given bit pump if a different bit pump causes an interrupt.   This allows other bit pumps to have interrupts serviced without affecting the one or more bit pumps that have their interrupts masked.

The ZipWire software does not define this directive because it has not been exhaustively tested. However, Conexant is aware of customers who are successfully using this directive.

C51

The C51 directive should be declared when the ZipWire software is executed on an Intel 80C32 family microprocessor and the Keil C compiler is used to compile the source code.

If another Intel 80C32 family compiler is used, the C51 directive can be defined. The user must be aware that there may be compatibility issues among compilers. If a non-Intel C51 family microprocessor and C compiler are used, do not define the C51 directive.

CHAN_UNIT

The CHAN_UNIT directive causes additional software to be compiled that supports the channel unit device. On the EVM, the channel unit resides on a separate board. Refer to *RS8953B Application and Channel Unit Software Developer's Guide* (100nnnx, formerly NCHUDG1B) for more details.

NOTE: The CHAN_UNIT directive must not be used when _NO_OF_LOOPS > 3 unless the destination field values for the API commands are modified (see Section 7.5).

ERLE

The ERLE directive enables the ERLE diagnostic code. Refer to Section 8.5 for details.

HTUR and HTUC

The HTUR and HTUC directives control whether HTU-R or HTU-C code is included in the executable. By defining both these directives, a single executable is created that supports both the HTU-R and HTU-C. In this situation, the terminal type is selected at run time.

If code space is limited, the HTU-R and HTU-C code can be customized for each terminal type. Because the bit pump ASM functions each represent about 9 kB of ROM, it is possible to save ROM space by only defining HTUR for the HTU-R code and HTUC for the HTU-C code. In this case, the _TERMINAL_TYPE command must match the terminal type selected at compile time. See Section 2.6 for more information.

INT_BUG

The INT_BUG directive must not be defined. The INT_BUG directive uses microprocessor timers instead of internal timers in the bit pump. This directive was needed in an earlier bit pump device.

NO_INDIRECT_RAM_VARS (added in ZipWire Software Version 3.0)

The NO_INDIRECT_RAM_VARS directive moves variables stored in internal RAM in the bit pump to external RAM. Customers with adequate external memory can declare this directive. The moved variables are declared in extern.h:

```
#ifdef NO_INDIRECT_RAM_VARS
   BP_S_8BIT eq_ram_vars[NUM_EQ_RAM_VARS][EQ_RAM_BYTE_WIDTH];
   BP_S_8BIT scratch_pad_vars[NUM_SCRATCH_PAD_VARS];
#endif
```

This directive increases the microcontroller execution speed because some, but not all, *Delay2Symbols()* function calls are eliminated. A 2-symbol delay is required after writing to any register in the bit pump indirect memory area. This delay is implemented in the ZipWire software with the *_Delay2Symbols()* function and is enabled with the ADD_DELAY directive.

This directive also assists software debugging because the variables can be examined using an emulator.

To verify if the NO_INDIRECT_RAM_VARS directive is working correctly, stop the emulator after executing this statement in *_BtInitialize()*:

```
WR_BYTE(no, STATUS, INIT_STATUS_REG); /* Reset status register */
```

where

```
#define INIT_STATUS_REG  0x03
```

If *eq_ram_vars[3][0]* equals 0x03, the directive is working correctly. The first array index (i.e., [3]) is computed as follows:

$$\text{STATUS} - \text{EQ\_RAM\_FIRST\_VAR} = 38 - 35 = 3$$

The NO_INDIRECT_RAM_VARS directive requires 82 bytes of RAM per bit pump (64 bytes for the scratch pad memory and 18 bytes for the equalizer RAM). In the Intel 80C32 environment, these variables are allocated in the BP_XDATA data space; therefore, the system must have external RAM.

When the NO_INDIRECT_RAM_VARS directive is not declared, the ZipWire software uses two places of extra memory in the bit pump to store variables. This extra RAM was needed in the original EVM which did not contain any external RAM. One of these areas is listed as reserved in the bit pump data sheet for the equalizer register (0x78).

The complete list of variables stored in the equalizer RAM area is found in bitpump.h starting after decimal 34:

```
#define STAGE            35
#define FCELL__SU_FLAGS  36
#define OPEN_ATTEMPTS    37
#define STATUS           38
#define USER_SETUP       39
#define PARAMETERS       40
#define FELM_THRESHOLD   41
#define DAGC_VALUE       42
#define ERLE_SETUP       42
#define STAGE2           43
```

Table 5-6 lists descriptions of the data stored in the equalizer RAM area.

Table 5-6. Data Stored in Equalizer RAM

RAM #	#define	High Byte	Low Byte
0 – 34	n/a	eq_add_read (0x78) registers	
35	STAGE	Test Mode State	Bit Pump ASM State
36	FCELL__SU_FLAGS	Startup Flags (*su_flags* local variable)	Frequency Cell Counter, HTU-R only
37	OPEN_ATTEMPTS	Unused	Number of Opened Phases (HTU-C) Number of Initial Open Eye Attempts (HTU-R)
38	STATUS	Unused	Startup Status (*status_reg* local variable)
39	USER_SETUP	User Setup, High Byte (See Table A-5)	User Setup, Low Byte (See Table A-5)
40	PARAMETERS	LOST Timer Value	Application Parameter, Low Byte (SeeTable A-7)
41	FELM_THRESHOLD	High FELM Threshold	Low FELM Threshold
42	DAGC_VALUE	During Temperature and Environment Handling Low Byte and High Byte = Previous DAGC Value, 16-bit	
	ERLE_SETUP	During ERLE	
		Unused	ERLE Setup
43	STAGE2	Bit Pump State after Waiting Meter Intervals	*_HandleTempEnv()* State
44–47	Reserved for Bit Pump Hardware		

The second RAM area is the scratch pad RAM located in registers 0x76 and 0x77 in the bit pump. The software macro WR_PHASE_QUALITY(i, temp_s16) in the code stores 16 bit values into the scratch pad area.

PDATA_MODE, XDATA_MODE

Either the PDATA_MODE or the XDATA_MODE directives must be declared with the C51 directive. The PDATA_MODE and XDATA_MODE directives set the addressing mode used to access the bit pump chips. Only one, not both, of these directives must be declared in a C51 system.

The PDATA_MODE directive selects paged addressing, which results in reduced ROM requirements for code space and faster execution. The XDATA_MODE directive selects external non-paged addressing, which results in greater code size and ROM requirements, and somewhat slower execution of the control code.

For PDATA_MODE, in each of the bit pump functions, the microcontroller P2 port is set to address the specified bit pump. In addition, the bit pump interrupt saves the P2 port value before addressing the bit pumps, then restores the previous P2 port value on exit of the interrupt function. Special care must be taken so that no other interrupt handler or code section erroneously modifies the P2 port value.

NOTE: If the Keil compiler’s Large Memory Model is defined, the XDATA_MODE directive should be set because all variables are declared in xdata space (see Section 5.10.4.2).

PRINTF_NOT_SUPPORTED (added in ZipWire Software Version 3.0)

Use the PRINTF_NOT_SUPPORTED flag in systems that do not support the *printf()* command. The *printf()* command is used with the TDEBUG directive to output debugging information during the activation process.

This directive replaces the *printf()* function with the *my_printf()* function found in user.c. The *my_printf()* function uses the *vsprintf()* function to copy the formatted string into *printf_buf[256]*. The *printf_buf[256]* character string is allocated in BP_XDATA space. The *put()* function then outputs the string. The *put()* function should then be replaced by whatever equivalent function is used in the desired system, e.g., *outp()* or *output()*.

SER_COM

The SER_COM directive enables the serial communication protocol of the user interface. It is used in the dual processor architecture. It should be declared in an 80C32-based system that employs the serial communication configuration for controlling the bit pump. The SER_COM directive can be declared only if the C51 directive is declared.

SINGLE_LOOP; TWO_LOOPS; THREE_LOOPS; FOUR_LOOPS; SIX_LOOPS

These directives allow the developer to specify the maximum number of loops supported in their system. The number of loops is equivalent to the number of bit pumps. Only one of these directives can be defined at a time. The bit pumps are allocated in a sequential order starting with _BIT_PUMP0 (the first bit pump) and ending with _BIT_PUMP5 (the sixth bit pump).

Only bit pumps that are supported by the specified flag can be accessed during run time, otherwise, unpredictable behavior results. For example, if THREE_LOOPS is defined, only _BIT_PUMP0, _BIT_PUMP1, and _BIT_PUMP2 can be accessed.

Each subsequent loop adds ~8 bytes of ROM space and ~4 bytes of RAM.

These directives support five options for the number of bit pumps. Version 4.0 modified the software to easily support any number of bit pumps. See Section 5.9.5 for details.

TDEBUG

The TDEBUG directive normally should not be declared. TDEBUG activates *printf()* statements that output diagnostic information (see Section 8.2). The messages are printed to the 80C32's serial port; therefore, TDEBUG cannot be used when SER_COM is declared.

TEMP_ENV

The TEMP_ENV directive compiles in the *_HandleTempEnv()* function that handles any temperature and environmental changes to the DSL. This code adds ~750 bytes of ROM. See Chapter 6.0 for details. This directive is defined in typedefs.h and therefore need not be declared in the project file.

ZIP_START (added in ZipWire Software Version 3.0)

This directive enables the compilation of the ZipStartup software (warm restart feature). See Section 2.8 for additional details.

5.8 Code Modifications

The code modifications required to support C51 and non-C51 environments are listed in this section. Additional code modifications that may be required for high-end applications (i.e., applications with many bit pumps or high-speed microprocessors) are discussed in Section 5.9.

The exact locations of code sections to be modified are designated by special comment blocks in the source files as follows:

```
   /*---------------------------------------------------------*/
   /*>>>   User Modifiable Section user.1 Begins          <<< */
   /*---------------------------------------------------------*/
   /* Modification Description:                             */
   /* Write the contents of routine EnableUserInterrupts().   */
   /* Calling this routine should enable all user interrupts. */
   /*                                                       */
   /* Reference:                                            */
   /* “ZipWire Software User's Manual”.                     */
   /*---------------------------------------------------------*/
Insert code here

   /*---------------------------------------------------------*/
   /*>>>   User Modifiable Section user.1 Ends             <<<*/
   /*---------------------------------------------------------*/
```

These comments contain basic information describing the code section that should be modified (or written). All modifications are referenced according to their source file name and the order in which they appear in the file. This reference (filename.c.n) appears both in the source file comments and in this manual.

Each of the following sections refers to a single code section that must be modified or written. All necessary modifications are covered, and no other changes in the source code should be made.

The bit pump source files that contain modifiable sections are btint.c, dsl_init.c, serint.c, btmain.c, typedefs.h, user.c, and user.h. Do not modify any other file.

5.8.1 btint.c.1

Code section btint.c.1 contains the header of *_BpInterruptHandler()*. The body of the interrupt function is already written and should not be modified in any way. However, the function header syntax is hardware-, microprocessor-, and compiler-specific and must be written for the specific configuration.

This function should be invoked each time a bit pump interrupt signal is asserted (the bit pump INT pin is a level-triggered, active-low signal). This interrupt should be enabled on power-up. It is necessary to ensure the interrupt vector is initialized in the application.

When using the Borland C compiler, the interrupt function header is defined as follows:

```
void interrupt _BpInterruptHandler(void)
```

If required, add code to the power-up initialization code to configure *_BpInterruptHandler()* as the bit pump interrupt function.

5.8.2 dsl_init.c.1

As discussed in Section 3.1, *_DSLInitialization()* calls *_Init8051()* to initialize the Intel 80C32 microprocessor. When using a different processor, the processor initialization function can be called in this portion of the software.

5.8.3 serint.c.1 (requires defining SER_COM)

Code section serint.c.1 contains the header of the serial interrupt handling function. The body of the interrupt function is already written. However, the function header syntax and some variables are hardware-, microprocessor-, and compiler-specific and must be written for the specific configuration.

This function should be invoked every time a serial interrupt signal is asserted. This interrupt should be enabled on power-up. It is necessary to ensure the interrupt vector is initialized in the application.

When using the Borland C compiler, the interrupt function header is defined as follows:

```
void interrupt Serial(void)
```

If required, add code to the power-up initialization code to configure *Serial()* as the serial interrupt function.

5.8.4 bt_main.c.1

The bit pump microcomputer interface works in either synchronous or asynchronous mode. Synchronous mode reduces the internal noise generated in the chip. However, in this mode, the access times on the 80C32 microcontroller fail the microprocessor timing specifications at data rates below ~250 Kbps.

The ZipWire software initializes the bit pump device in the asynchronous mode. Lab tests have not shown any performance degradation when running in this mode.

For host microprocessors that require the DTACK input, the bit pump device must be configured for the synchronous mode. In this mode, the READY output from the bit pump is valid, which can be used as DTACK input for the microprocessor. To configure the bit pump for synchronous mode, modify the following line of code:

Change:

```
BP_WRITE_REG(bp_ptr, reserved9, ASYNCHRONOUS_MODE);
```

to

```
BP_WRITE_REG(bp_ptr, reserved9, SYNCHRONOUS_MODE);
```

5.8.5 typedefs.h.1

Code section typedefs.h contains the equivalent data type definition to match the compiler. Table 5-7 lists the bit pump data types that must be specified.

Table 5-7. Bit Pump Data Types

Bit Pump Data Type	Description	Keil Equivalent(1)	GNU C Equivalent(2)
BP_BIT_FIELD	1-bit field	unsigned char	unsigned char
BP_S_8BIT	signed 8-bit	char	char
BP_U_8BIT	unsigned 8-bit	unsigned char	unsigned char
BP_S_16BIT	signed 16-bit	short	short
BP_U_16BIT	unsigned 16-bit	unsigned short	unsigned short
BP_S_32BIT	signed 32-bit	long	int
BP_U_32BIT	unsigned 32-bit	unsigned long	unsigned int
BP_TABLE	signed 16-bit	short	short
BP_CONSTANT	constant	code	const
BP_VOLATILE	volatile	volatile	volatile
BP_DATA	Internal Memory	data(3)	N/A
BP_IDATA	Indirect Internal Memory	idata(3)	N/A
BP_PDATA	External Memory	pdata(3)	N/A
BP_XDATA	External Memory	xdata(3)	N/A

NOTE(S):
(1) The bit pump code is targeted towards the Keil compiler.
(2) Assuming 32-bit GNU C compiler.
(3) The data, idata, pdata, and xdata types are unique to the Keil 80C32 compiler.

5.8.6 user.c.1

Code section user.c.1 contains the absolute I/O address definitions of the specified number of bit pump chips as defined by the _NO_OF_LOOPS flag. Any value can be defined for non-existent bit pump chips.

The address value depends on the address-decoding scheme implemented in the system. The defined address should reflect the value that accesses register 0 on the bit pump.

5.8.7 user.c.2

Code section user.c.2 contains two noise margin calibration tables. One table is calibrated for ANSI ISDN loop 4; the other is calibrated for ETSI loop 2 with shaped noise and low crest factor. The ZipWire software uses the ETSI calibrated table as the default. If the actual noise source is an ANSI loop, the ETSI-calibrated table provides an NMR reading that is higher than actual. The higher reading avoids a false deactivation by the application ASM because of a low NMR. As discussed in Sections 3.9.6 and 3.10.6, the ASM should use other, more accurate readings to detect a bad DSL. The noise margin table can be calibrated for another noise source by performing the calibration procedure provided in Appendix B:.

5.8.8 user.c.3

Code section user.c.3 contains code that initializes the array of pointers used to access the bit pump chips. The software stores the bit pump addresses in the *_bit_pump[_NO_OF_LOOPS]* array. This array is declared and initialized using the software macros listed in Table 5-8.

Table 5-8. _bit_pump[] Array Declaration and Initialization

Array	Variable Declaration	Variable Initialization
_bit_pump[_NO_OF_LOOPS]	DECLARE_PTR_ARRAY	*_BtHomerPointersInit()*

5.8.9 user.c.4

The contents of the *_Delay2Symbols()* function are application-specific. This function must implement a delay of 2 symbol periods. Accesses to an indirect register in the bit pump require a 2-symbol delay after writing to the register. Indirect registers have addresses greater than 0x70 and are accessed indirectly by reading and writing to intermediate registers (called access data registers).

The delay is implemented in the software using the *_Delay2Symbols()* function. Some of the *_Delay2Symbols()* calls in the software are not compiled when the NO_INDIRECT_RAM_VARS directive is defined.

NOTE: Depending on the microprocessor and compiler, entering and exiting this function can generate the required delay with no need for additional operations or NOPs.

Use these steps to verify that *_Delay2Symbols()* provides adequate delay:

1. Write test code that implements this pseudo-code:

```
while(1)
 {
  set port line high
  _Delay2Symbols();
  set port line low
 }
```

2. Use an oscilloscope to measure how long the port line remains high.
3. Ensure the measured value in Step 2 equals or exceeds the 2-symbol time.

The 2-symbol time can be computed using the equations provided in Section 4.3.

5.8.10 user.c.5

Code section user.c.5 contains the body of *_EnableUserInterrupts()*. When this function is executed, it enables all the non-bit pump interrupts disabled by *_DisableUserInterrupts()* (see user.c.6).

5.8.11 user.c.6

Code section user.c.6 contains the body of the function *_DisableUserInterrupts()*. The ZipWire software on the HTU-R must be executed 4 ms after the bit pump interrupt occurs in critical activation phases. When this function is executed, it disables all non-bit pump interrupts that exceed the 4 ms time criteria. The *_DisableUserInterrupts()* and *_EnableUserInterrupts()* functions are called by the ZipWire software in critical activation phases. On the EVM, the ZipWire software disables the serial interrupt. See Section 6.4 for more details.

5.8.12 user.c.7

Code section user.c.7 contains the body of the function contents *_WaitMicroSecond()*. This function must implement a minimum delay of 1 µs. *_BitpumpSetSymbolRate()* calls *_WaitMicroSecond()* after it configures the PLL in the bit pump. After writing to the bit pump PLL modes register (0x22, bits [7:6]), a delay of at least 200 µs is required to ensure that the PLL is stable. During this time, access to the bit pump device is not permitted.

5.8.13 user.h.1

Code section user.h.1 contains the body of *_EnableBitpumpInterrupt()* and *_DisableBitpumpInterrupt()*. These functions are used in two scenarios.

The first scenario occurs when performing a read operation on the bit pump indirect RAM (i.e., EQ RAM, LEC, etc.). During the read operation, the bit pump cannot tolerate any other microprocessor write operation until the data is read from the access data byte (0x7C–0x7F) registers. This includes the 2-symbol delay needed for the data to go from the indirect RAM to the access data registers.

In the second scenario, when reading a meter register (i.e., SLM, FELM, BER, etc.), the bit pump must read the low byte followed by the high byte with no intermediate meter access. When the BER meter is enabled (which reads the ber_meter registers [0x4C and 0x4D] in the interrupt handler every meter interval), any other meter read in between reading the low and high meter byte is corrupted by the interrupt handler ber_meter access.

The *_DisableBitpumpInterrupt()* and *_EnableBitpumpInterrupt()* macros in user.h disable and enable the bit pump interrupt to prevent the interrupt handler from corrupting data in either of these two scenarios. The bit pump *ReadAccessDataByte()* function and READ_METER_REG() macro call the *_DisableBitpumpInterrupt()* before performing the bit pump access, then call *_EnableBitpumpInterrupt()* on completion.

5.9 Additional Modifications for High End Applications

High-end applications use 16- or 32-bit microprocessors or multiple bit pumps. The ZipWire software has been successfully ported to these applications on numerous occasions. This section provides guidelines on porting the ZipWire software to these high-end applications.

5.9.1 RAM and ROM Requirements

When porting the bit pump code for a 16- or 32-bit microprocessor, RAM and ROM requirements are typically not critical. When the C51 directive is not defined, the software leaves the BP_DATA, BP_IDATA, BP_MSPACE, etc., memory spaces blank and uses the default memory space specified by the compiler.

5.9.2 Bit Pump Memory Accesses

The BP_WRITE_REG(), BP_READ_REG(), BP_WRITE_BIT(), and BP_READ_BIT() macros provide access to the bit pump registers. In environments that cannot easily interface with the bit pump device, these macros can be modified.

The BP_WRITE_REG() and BP_READ_REG() macros access the bit pump registers at a byte level. The BP_WRITE_BIT() and BP_READ_BIT() macros access the bit pump registers at a bit-field level. These macros are found in bitpump.h.

High-speed processors running at low data rates can fail to meet the bit pump internal register hold time requirement when performing two consecutive BP_WRITE_REG() operations. The internal write timing requirements are specified in *RS8973 Single-Chip SDSL/HDSL Transceiver* (100nnnx, formerly N8973DSD) Figure 5-10. For example, at a data rate of 160 Kbps, a delay of 400 ns must be added to the BP_WRITE_REG() macro. One way to add this delay is illustrated in the following pseudo-code:

```
BP_U_8BIT some_func(BP_U_8BIT value);
   #define BP_WRITE_REG(ptr, reg, value)   ptr->reg = some_func(value)

BP_U_8BIT some_func (BP_U_8BIT reg_value)
   {
      add_400ns_delay();
      return reg_value;
   }
```

The *_SelfTest()* function uses the SET_WORD() macro that performs two consecutive BP_WRITE_REG() operations. Therefore, if the bit pump internal register hold time is violated, the *_SelfTest()* function returns a value of FAIL.

Two consecutive BP_WRITE_BIT() operations can also violate the bit pump internal register hold time requirement. Because the BP_WRITE_BIT() macro performs a read-modify operation before performing the write operation, it has a built-in delay and can require less of a delay than the BP_WRITE_REG() macro or no additional delay.

5.9.3 Bit Pump Global Variables

Customers designing systems with multiple bit pumps may wish to dynamically allocate memory to individual bit pumps as they are enabled and disabled. Table 5-9 lists three important global arrays. The largest structure that contains bit pump variables is *_bp_vars[]* array. For each bit pump, the *_bp_vars[]* array allocates 96 bytes of memory when the BER_METER and NO_INDIRECT_RAM_VARS flags are defined.

Table 5-9. Bit Pump Global Variables

Array	Array Definition	Array Declaration
_bp_vars[_NO_OF_LOOPS]	extern.h	btmain.c
_int_reg[_NO_OF_LOOPS]	ptrdef.h and bitpump.h	DECLARE_INT_REG in btint.c
_bit_pump[_NO_OF_LOOPS]	ptrdef.h and bitpump.h	DECLARE_PTR_ARRAY in user.c

5.9.4 Bit Pump Local Variables

The software uses software macros when declaring and initializing local variables that access the bit pump. The macros for variable declarations, such as DECLARE_PTR, are located at the top of a function where local variables are allocated. The macros for variable initialization, such as INIT_BP_PTR, are located in the function body and are called if a read or write access to the bit pump is required.

Table 5-10 lists the local pointer variables and the macros that declare and initialize them.

Table 5-10. Local Pointer Variables to the Bit Pump Device

Pointer Variable	Variable Declaration	Variable Initialization
**bp_ptr*	DECLARE_PTR	INIT_BP_PTR
**bp_mode_ptr*	DECLARE_MODE_PTR	INIT_BP_MODE_PTR
**int_ptr*	DECLARE_INT_PTR	INIT_INT_PTR

The **bp_mode_ptr* and **bp_ptr* variables both point to the same bit pump address. However, the **bp_mode_ptr* variable only points to registers that have been defined with bit fields. This distinction allows the Keil compiler to optimize accesses to bit fields. The **bp_mode_ptr* variable is passed to the BP_WRITE_BIT() macro, and **bp_ptr* variable is passed to the BP_WRITE_REG() macro.

5.9.5 Setting the _NO_OF_LOOPS Directive

The SINGLE_LOOP, TWO_LOOPS, THREE_LOOPS, FOUR_LOOPS, and SIX_LOOPS directives can be used to automatically define the _NO_OF_LOOPS directive. Alternatively, the _NO_OF_LOOPS can be manually defined in btmain.h if the number of bit pumps exceeds six.

In addition, the _NO_OF_LOOPS and the *_bit_pump[]* array can be set based on a product directive. Using a directive makes it easier to manage the code for multiple products.

For example, the ZipWire software uses the ZipSocket directive to support the ZipSocket product. Refer to btmain.h:

```
#ifdef ZIPSOCKET
/* Zip Socket Module Product */
#define _NO_OF_LOOPS 2
#else
/* EVM */
#define _NO_OF_LOOPS 3
#endif
```

Also, refer to user.c:

```
static BP_U_16BIT BP_CONSTANT bit_pump_addr[] =
{
#ifdef ZIPSOCKET
  0x8000
  0xA000
#else
  0xF000
  0xD000
  0xE000
  0xF000
  0xF000
  0xF000
endif /* ZIPSOCKET */
};
```

NOTE: For the EVM software, the *bit_pump_addr[]* array has 6 entries to maintain backward compatibility with the SIX_LOOPS compiler directive.

5.9.6 PACK Pragma

The pragma at the top of bitpump.h:

```
#pragma pack(1)
```

is typically required by customers using Motorola processors. Most C compilers, such as GnuC, support a packing option that compresses multiple bytes into one word. This option is enabled by a *#pragma pack()* or by a command line option such as /Z. If a compiler places padding in between bytes in a structure, the packing option must be enabled to eliminate this padding. In bitpump.h, the bit pump registers are defined in the transceiver structure. The bit pump software assumes there is no padding in the structure. The byte-addressing mode feature of Motorola processors ensures that the correct packed byte is accessed.

Set up the pack pragma by defining BP_PACKED in typedefs.h:

```
#define BP_PACKED    __packed__(1,1,0)
```

or removing the comment around /* #pragma pack(1) */ at the top of bitpump.h and modifying the parameter options, if needed:

```
#pragma pack(1,1,0)
```

The *pack()* parameter options can vary among compilers.

The BP_PACKED2 directive is used for compilers that require the pack pragma to follow the structure definition. The BP_PACKED2 directive is shown in this example:

```
typedef BP_PACKED struct{
    BP_BIT_FIELD hclk_freq:2;
    BP_BIT_FIELD smon_source:6;
} BP_PACKED2 serial_monitor_type;
```

5.10 Special Issues for the 80C32 Microcontroller

There are four special issues for the 80C32 microcontroller.

5.10.1 Memory Storage Options

The 80C32 microprocessor has several locations in which to store variables: data, idata, xdata, etc. When selecting these spaces, care must be taken to understand the RAM and ROM requirements for different options.

The data space is fastest and requires the least amount of ROM. However, data space is limited to 128 bytes. The idata space is slightly slower and requires slightly more ROM than the data space. The idata space is limited to 128 bytes.

NOTE: The idata space can reside in the data space if there is adequate space. The data space cannot reside in the idata space.

The xdata space is slower and requires a significant amount of ROM space because the 8-bit microprocessor has to push around 16 bits of data. However, the xdata space can be quite large (up to 64 KB).

The 256 bytes of the data and idata space also include the bdata (bit field), DATA_GROUP, and stack. The bdata space could potentially create gaps in the data space as the linker tries to optimize the RAM usage. Table 5-11 demonstrates the impact of placing the *bp_vars[]* array into different memory spaces.

Table 5-11. Location of bp_vars[] versus Memory Sizes

Memory Space	RAM	RAM GAP	XDATA	ROM
data	0xC9	0	0	23,204 KB
idata	0xD2	0x09	0	23,251 KB
xdata	0xBC	0x13	0x22	25,424 KB
NOTE(S): These results show one simple compiler option and vary based on the software release. The ZipWire software places the *bp_vars[]* array in the xdata space.				

5.10.2 Selecting Storage Options for Variables

The ZipWire software explicitly declares the memory location for variables, which allows them to be allocated in either the data, idata, or xdata memory locations. Software flags with names ending in _MSPACE make it easy to change the memory location for blocks of variables. The flags are listed in Table 5-12.

Table 5-12. Flags Used to Allocate Blocks of Variables

Flag Name	Block of Variables	Default Memory Location	Location of #define
BP_MSPACE	bit pump variables	xdata	typedefs.h
APPL_SW_MSPACE	application variables	xdata	typedefs.h
TIMER_MSPACE	timer variables	idata	timer.c
MAIL_MSPACE (requires defining SER_COM)	serial comm. mailbox	xdata	mail.c
ZIP_START_MSPACE (requires defining ZIP_START)	ZipStartup coefficients	xdata	zipstart.h

5.10.3 ROM Options

When using the Keil compiler, the ROM option of LARGE must be used. Because the bit pump code contains functions that are larger than 2 KB, the COMPACT ROM option cannot be used.

5.10.4 Memory Model Options

The ZipWire software is compiled with the default option of a small memory model. However, Conexant is not aware of any problems when specifying a large memory model.

5.10.4.1 Small Memory Model

For a small memory model, either the PDATA_MODE or XDATA_MODE directive must be defined. Because the bit pump registers are mapped to external memory, these directives specify how to access external memory. If the external memory requirements are less than 256 bytes, the PDATA_MODE directive is preferred because it creates less and faster code. For a small memory model, the compiler locates bit pump static variables to the default location of data which is the 256 bytes of internal RAM in the 80C32.

5.10.4.2 Large Memory Model

When using a large memory model, the XDATA_MODE directive must be defined. In this case, the compiler will locate static variables to xdata which is in external RAM. The PDATA_MODE directive should not be used for the large memory model. The large memory model is not as efficient as the small memory model because it does not use the on-board RAM for storing bit pump static variables.

6.0 Real-Time Constraints

In HDSL systems that employ single processor architecture, the application software is integrated with the ZipWire software. Because the application and ZipWire tasks share the same processor, care must be taken when integrating the two tasks to avoid degrading the bit pump performance. This chapter outlines the real-time constraints to consider when integrating the ZipWire and the application software.

6.1 Overview

All ZipWire software operations, including activation control, monitoring, and normal operation activities, are performed only when the *_BtMain()* function is called by the main program. During the time an application task occupies the processor and prevents *_BtMain()* from being executed, ZipWire tasks are delayed. The real-time constraints for a system depend on the system configuration and which tasks are being handled. The most notable system dependencies are:

- Activation Process versus Normal Operation
- HTU–C versus HTU–R

During certain activation phases, such a delay may cause symptoms such as longer activation time or even activation failure; however, the added delay does not have a significant impact on the final noise-margin performance. During normal operation, such delays are less critical but may still degrade the performance in certain situations, such as environmental changes.

6.2 Activation Process

During the activation process, only the *_HandleFlags()*, *_HturControlProcess()*, and *_HtucControlProcess()* functions of *_BtMain()* have any impact on the real-time constraints. The *_HandleTempEnv()* function performs no action during the activation process. There are four critical factors to consider when implementing the system: microprocessor demand, activation reliability, activation times, and final noise performance.

6.3 NO_INDIRECT_RAM_VARS Directive

There are many bit pump variables (i.e., state number, configuration, etc.) stored in the bit pump's scratch pad and equalizer RAM. Access to these indirect RAM variables are slow because each access requires a 2-symbol delay. The NO_INDIRECT_RAM_VARS directive allocates these bit pump variables in RAM instead of the bit pump indirect RAM. This dramatically decreases access times during *CalculateOptimalPhase()* and other functions that use the indirect RAM.

The NO_INDIRECT_RAM_VARS directive requires 82 bytes of RAM per bit pump (64 for scratch pad and 18 for equalizer RAM). In the Intel 80C32 environment, these variables are allocated in BP_XDATA space, and therefore the system must have external RAM.

6.4 Microprocessor Demands

The execution time for each bit pump activation state is less than 2 ms except for the long states listed in Table 6-1 for the HTU-C and Table 6-2 for the HTU-R. The execution time includes symbol delays and microprocessor execution time and is dependent on the data rate.

Table 6-1. Execution Times for Long Functions for the HTU-C (1,168 Kbps data rate)

State	Function	Execution Time with Indirect RAM Time (ms)	Execution Time without Indirect RAM Time (ms)
OPEN_EYE6	_CalculateOptimalPhase1()	20	6.5
CALC_OPT_PHASE2	_CalculateOptimalPhase2()	8	2.3
CALC_OPT_PHASE3	_CalculateOptimalPhase3()	33	13.5
CALC_OPT_PHASE4	_CalculateOptimalPhase4()	35	11.0
CALC_OPT_PHASE	_CalculateOptimalPhase()	7	2.3

On the HTU-C, the *_CalculateOptimalPhase1()* function and various derivatives of it perform several search iterations to determine the best phase quality measurement based on the phase quality and NMR. The phase quality and NMR results are stored in the Scratch Pad RAM which requires a 2-symbol delay for each read and write access. As listed in Table 6-1, defining the NO_INDIRECT_RAM_VARS flag greatly reduces the execution time.

Table 6-2. Parameters for Long States for the HTU-R (1,168 Kbps data rate)

State	Execution Time (ms)
PHASE_LOCKED	7
OPTIMAL_PHASE	1,700
TRANSMIT_2LEVEL	500

6.4.1 WAIT_FOR_TIMER Software Macro

For the long states shown in Table 6-2, the HTU-R software executes a series of idle loops waiting for the expiration of a timer, called the meter timer, to expire in the bit pump. The timer runs while a bit pump meter is collecting data. When the timer expires, the bit pump interrupt is processed, and it clears a timer flag.

The WAIT_FOR_TIMER software macro implements the idle loop. The states listed in Table 6-2 call a series of WAIT_FOR_TIMER macros. The execution time is the time for the complete state processing including all WAIT_FOR_TIMER macros.

During the OPTIMAL_PHASE and TRANSMIT_2LEVEL states, the PLL in the bit pump is frozen. To minimize drift, the PLL should be unfrozen as soon as possible.

If a Real-Time Operating System (RTOS) is used, modifying the WAIT_FOR_TIMER macro can eliminate the wasted time in the idle loops. In the WAIT_FOR_TIMER macro, add a RTOS call to block the bit pump task until the bit pump interrupt occurs. This block is typically implemented by a RTOS call that creates a binary semaphore. When the bit pump task is blocked, it is context-switched to a “Wait for Event” state by the RTOS. The bit pump interrupt unblocks the bit pump task by releasing the semaphore. The RTOS then switches the task back to a “Ready” or “Running” state.

Some customers create the semaphore in the RESTART macro. However, creating the semaphore in the WAIT_FOR_TIMER macro is a more robust solution.

6.4.2 HTU-R Real-Time Requirements

For the long states shown in Table 6-2, the ZipWire software on the HTU-R must be executed within 4 ms after the meter timer expires. This requirement applies to any application or interrupt software running during the long states. The application software is customer software written in place of the WAIT_FOR_TIMER macro.

To ensure that user interrupts, such as the serial interrupt, do not violate this requirement, the ZipWire software disables user interrupts during the long states by calling *_DisableUserInterrupts()*. The *_EnableUserInterrupts()* function reenables all interrupts that were disabled by the *_DisableUserInterrupts()* call. The ZipWire software that uses a channel unit does not violate the 4 ms requirement; therefore, the channel unit interrupt need not be disabled.

NOTE: The bit pump interrupts must never be disabled for extended periods of time.

6.5 Activation Reliability, Times, and Performance

During the bit pump activation procedure, the bit pump can tolerate long delays between calls to *_BtMain()* without impacting the system performance. However, increasing the delay causes the activation time to increase. The HTU–C is extremely sensitive to delays between calls to *_BtMain()* as compared to the HTU–R. Table 6-3 lists the added activation time for HTU–C and HTU–R systems when there is a continuous delay between calls to *_BtMain()*.

Table 6-3. Added Activation Times in Seconds (784 Kbps system)

Delay (ms)	HTU–C	HTU–R
10	1	0
50	4	0
100	9	1
200	24	2
500	70	7

The added activation time does not cause the bit pump activation process to fail. However, the added time could cause application-level time requirements to fail, for example, the ETSI/ANSI HDSL 30-second activation time.

6.6 Normal Operation

During the bit pump's normal operation in a single processor architecture, only the *_HandleFlags()* and *_HandleTempEnv()* functions of *_BtMain()* have any impact on the real-time constraints. The *_HturControlProcess()* and *_HtucControlProcess()* functions perform no actions and thus add no significant microprocessor demands.

6.7 _HandleFlags() Function

The *_HandleFlags()* function polls the interrupt register and updates the bit pump STATUS register. The execution time is less than 1 ms. Because this function updates the bit pump STATUS register, it is required to call the _STATUS request API command (0x85) only once after each call to *_BtMain()*.

6.8 _HandleTempEnv() Function

The *_HandleTempEnv()* function temporarily adapts some of the equalizer filters to handle any temperature or environmental changes on the DSL. The filters are only adapted when the meter interval expires based on a user-defined number of symbols.

Table 6-4 lists the maximum temperature deviation when calling *_HandleTempEnv()* for different intervals.

- Meter intervals: number of meter intervals between calls to *_HandleTempEnv()*
- Period (time): equivalent time on a 784 Kbps system
- Maximum temperature change the system can tolerate in °C/hour

Table 6-4. _HandleTempEnv() Tolerances

Meter Intervals	Period (Time)	Maximum Temperature Deviation
1	84 ms	30 °C/Hour
2	168 ms	15 °C/Hour
3	252 ms	10 °C/Hour
6	504 ms	5 °C/Hour
12	1 sec	2.5 °C/Hour

The ZipWire software calls *_HandleTempEnv()* once every 98,304 symbols, which compensates for a maximum temperature variation of 10 °C/hour. This number of symbols equates to 252 ms at 784 Kbps. Therefore, it is only necessary to call *_HandleTempEnv()* a minimum of once every 98,304 symbols to provide an adequate tolerance to temperature and environmental changes.

The customer can change the maximum compensated temperature by changing the following #define in util.c:

```
#define DEFAULT_TE_METER_INTERVAL    3
```

This value determines the number of 32,768 meter intervals that must expire before *_HandleTempEnv()* performs temperature and environment processing. On every 32,768 symbol interrupt, *_HandleTempEnv()* increments a counter and compares it to a variable that is set to DEFAULT_TE_METER_INTERVAL.

6.9 Bit Pump Interrupt

_BpInterruptHandler() is part of the ZipWire software. Its header may be modified, but its contents should not be altered in any way.

The application code should quickly service the bit pump interrupt. Long interrupt response times could degrade bit pump performance. The bit pump interrupt should be enabled on power-up and should never be disabled by the application code except for brief periods of time when accessing indirect or meter registers in the bit pump.

6.10 API Response Times

The time to execute the *_BtControl()* and *_BtStatus()* functions takes less than 1 ms, except for the API commands listed in Table 6-5.

Table 6-5. API Response Times

API Commands	Number of Symbols	Time (ms) Symbol Delay	Total Time (ms)
_SYSTEM_CONFIG	58,000	148	153
_SYM_RATE	58,000	148	153
_RESET_SYSTEM	58,000	148	153
_TEST_MODE: Exit Test Mode	29,000	74	89
_TEST_MODE: Analog Loopbacks	—	—	3.5

6.11 Power-up Operations Sequence

The application software should ensure that a proper initialization procedure occurs at power-up or microprocessor reset condition.

The following operations must be performed prior to any activation of a ZipWire software function (either *_BtMain()* or an API function):

1. Call *_BtSwPowerUp()* to initialize bit pump software parameters
2. Call *_MaskBtHomerInt()* to prevent false interrupts
3. Initialize processor (or interrupt controller), and enable bit pump interrupts
4. Call *_BtMain()* repeatedly after initialization

7.0 API Overview and Serial Communications Interface

This chapter describes the Application Program Interface (API) portion of the ZipWire software. The API command set allows the system designer to have full control of all ZipWire features and to get status and performance monitoring information from the bit pump.

The interaction between the application software and the ZipWire software can be achieved in one of two configurations, depending on whether the application software is implemented on the same microprocessor as the ZipWire control software or on a different microprocessor. When only one microprocessor is used in a system, the application interfaces to the bit pumps using a code-level API. In a design in which the ZipWire software runs on an 80C32 microprocessor, and a second host microprocessor is used for the application software, a serial communication protocol between the two processors controls the bit pumps.

Figure 7-1 and Figure 7-2 illustrate the two possible control architectures.

Figure 7-1. Dual Processor Architecture

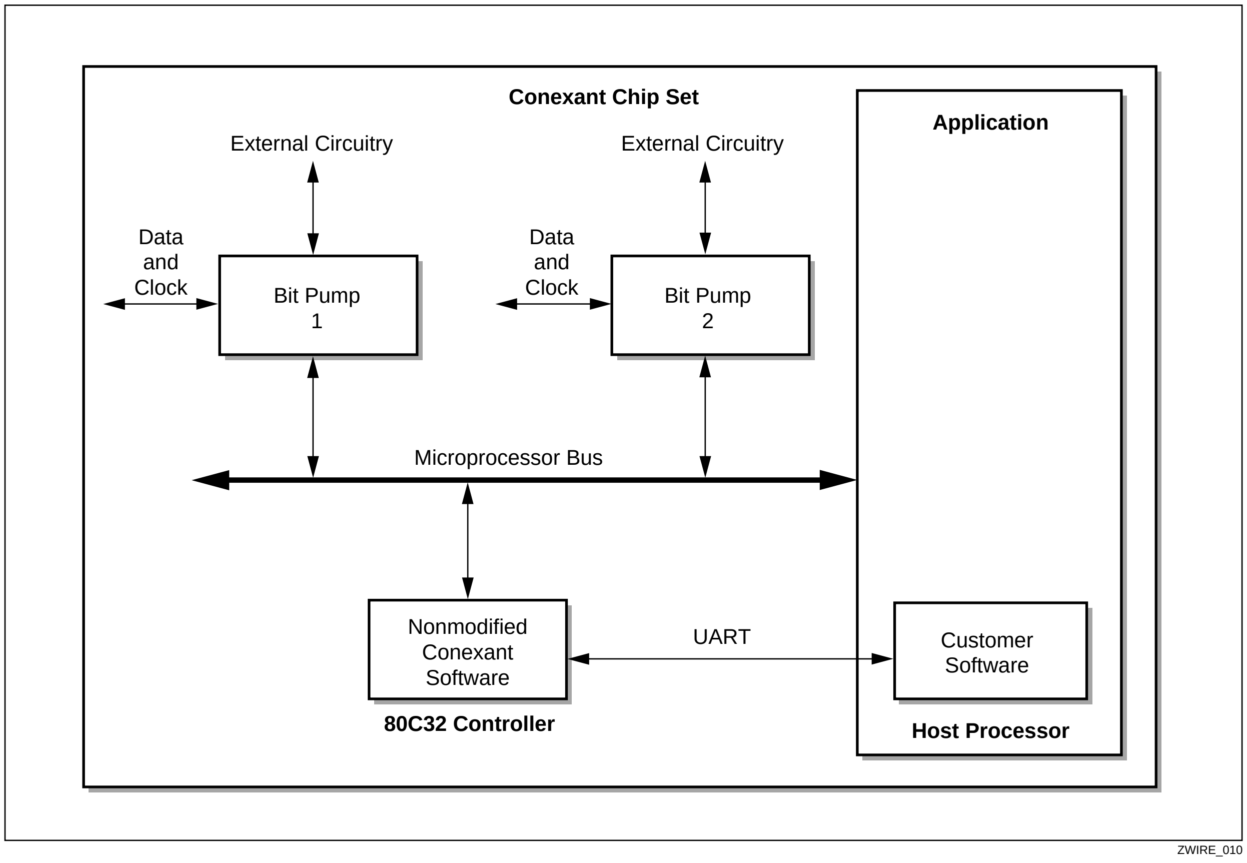
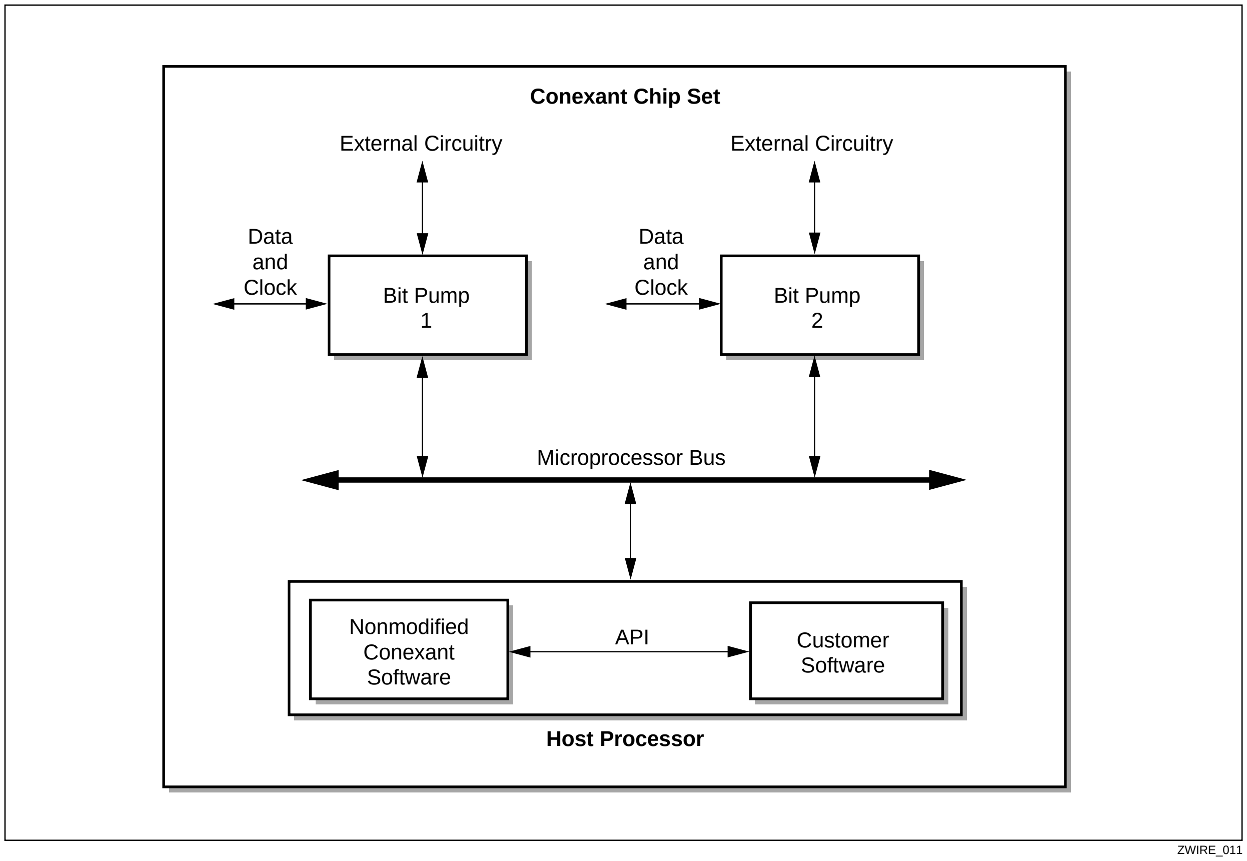


Figure 7-2. Single Processor Architecture



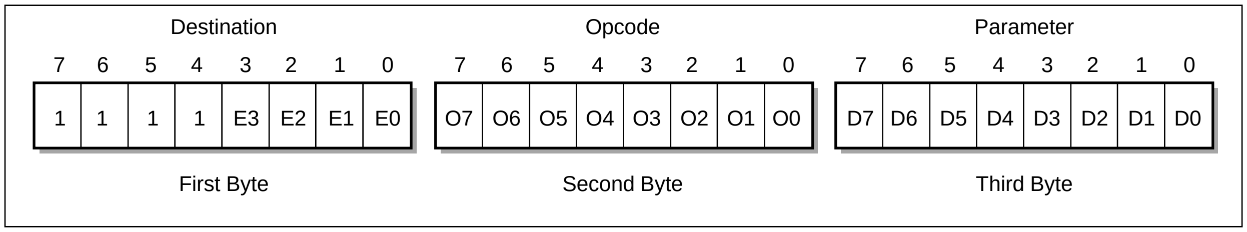
In both configurations, the control and monitoring operations are based on a set of API commands sent by the application to the bit pump software and a parameter status returned to the application. The API command set is identical in both configurations.

Section 7.1 describes the structure of commands and status responses, which applies to both configurations. Section 7.2 describes the serial interface message structure and communication protocol which applies to a dual processor architecture. Section 7.3 describes the API which applies to a single processor architecture. Section 7.6 lists the API commands. Appendix A: details the API command set including command syntax, operation, and application hints.

7.1 Command Structure

All commands have the same structure. A command is composed of three 1-byte fields: Destination, Opcode, and Parameter. Figure 7-3 illustrates the structure of a command.

Figure 7-3. Command Structure



ZWIRE_012

The command fields are interpreted as follows:

- Destination field (bits E3–E0)—This field selects the destination to which the command is targeted, as listed in Table 7-1. In a single processor configuration, the entire destination byte is available; the command can access up to 256 bit pumps.   In a dual processor configuration, the destination parameter is limited to 16 bit pump devices. This limitation allows backward compatibility with the existing serial communication protocol.

Table 7-1. Destination Field Specification

E3	E2	E1	E0	Destination
0	0	0	0	Bit Pump 0
0	0	0	1	Bit Pump 1
0	0	1	0	Bit Pump 2
0	0	1	1	Bit Pump 3
0	1	0	0	Bit Pump 4
0	1	0	1	Bit Pump 5

- Opcode field (bits O7–O0)—The Opcode field selects the specific command or status request to be executed. The available commands and their opcodes are described in detail in Appendix A:. The opcodes are also available in the C source file, api.h, which contains C constant definitions for all opcodes. Section 7.6 lists these opcode constants.
- Parameter field (bits D7–D0)—The parameters field is used in some commands where additional data or parameter selection is required. In commands where there is no need for additional data, 0s should be placed as the data byte to ensure future compatibility. The parameter field options are also available in the C source file, api.h, which contains C constant definitions for the available parameters. Section 7.6 lists these parameter field constants.

The set of commands is divided into three logical groups: control commands, status request commands, and the acknowledge message. The first group includes commands that control the bit pump operation and change parameters. The second group includes commands that request status values or monitoring information from the bit pump. In response to a status request command, a 1-byte data word containing the status information is returned to the application. The third command is the acknowledge message, which applies only to the UIP in a serial communication mode.

7.2 Serial Communication Interface

In a dual processor architecture (see Figure 7-1), the application uses a serial communication message transfer protocol to control the bit pumps. The protocol is based on the command set described in Appendix A: and the command structure as described in Section 7.1, with additional requirements for this type of interface.

7.2.1 Communication Protocol

The 80C32 microcontroller communicates with the host processor using a standard UART interface. The physical connection includes two lines: RXD (80C32 pin 10) and TXD (80C32 pin 11). The data is transferred in an asynchronous format: 9,600 baud, 1 start bit, 8 data bits, 1 stop bit, and no parity.

7.2.2 Message Transfer Protocol

The application sends a command to any bit pump in the system by transmitting a message over the serial communication channel. Every command that is correctly received and decoded by the 80C32 is acknowledged by returning the acknowledge message (see Section A.3 in Appendix A:) to the application. In response to a status request command, the 80C32 also sends a status response message that contains the requested information.

The 80C32 acknowledges a received message within 200 ms except during activation, where larger delays (up to 2 seconds) may be present. The host processor retransmits a message that was not acknowledged within this time limit. The host processor should not send a new message until the previous one is acknowledged, unless the time limit has been exceeded. A status request message requiring information from the bit pump is acknowledged, and only then is a response message containing the requested information sent to the host processor.

7.2.3 Message Structure

All messages are 4 bytes long. Figure 7-4 illustrates the structure of a message sent by the host processor to the 80C32. The first 3 bytes are the command bytes, as described in Section 7.1. The fourth byte (the last transmitted byte) contains checksum information that is a function of the first 3 bytes (see Section 7.2.4 for checksum function details). The checksum considerably reduces the probability of the 80C32 misinterpreting an incoming message.

When the 80C32 receives a status request command, it responds (after acknowledging the command) by sending a status message to the host processor. The structure of a status message is illustrated in Figure 7-5.

The first 2 bytes are identical to the first 2 bytes of the corresponding status request command. Bits S3–S0 of the first byte are interpreted, according to Table 7-1, as the source bit pump for the status response. The second opcode byte (bits O7–O0) contains the opcode of the command that requested the information. Bit O7 is set for a status-type command; it is cleared for a control-type command.

The third byte (bits D7–D0) contains the requested information. The fourth byte (bits CS7–CS0) is the checksum value, calculated according to the formula described in Section 7.2.4.

Figure 7-4. Host Processor to 80C32 Message Structure

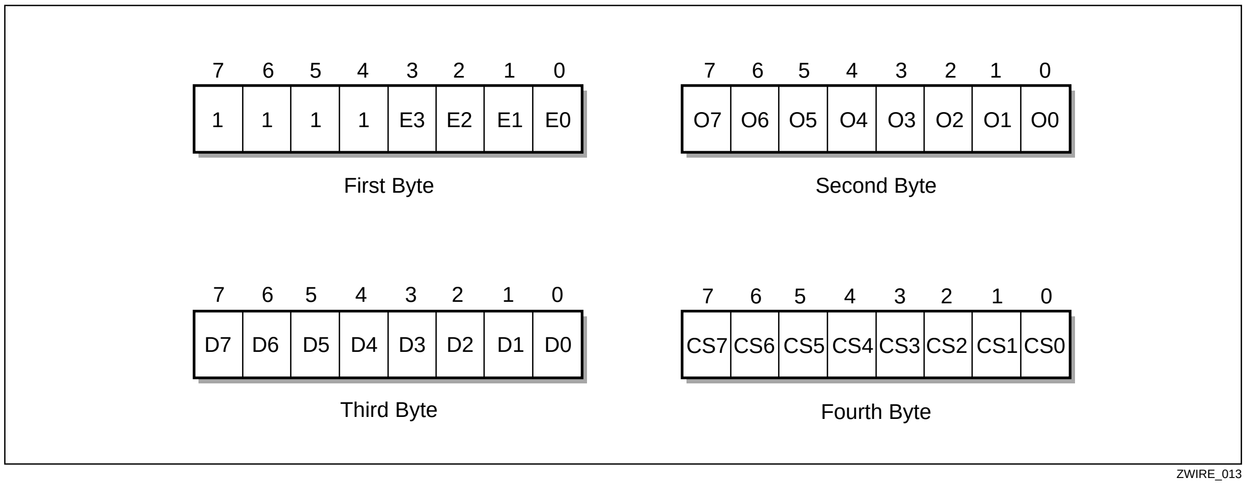
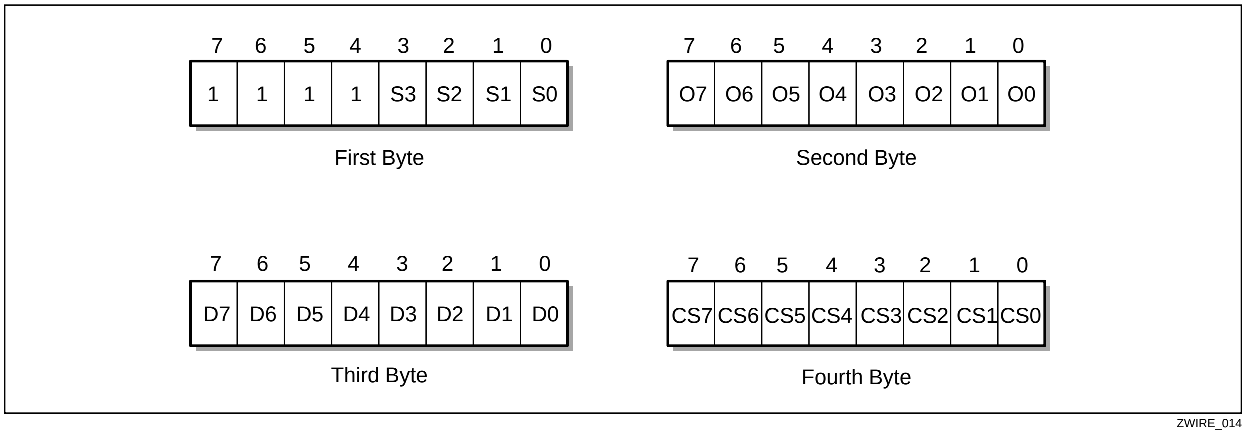


Figure 7-5. 80C32 to Host Processor Message Structure



7.2.4 Checksum Function

For every command sent by the host processor, a checksum function value is calculated and sent as the fourth byte of the message. This value is calculated using the following formula:

$$CS = (\text{Byte } 1) \oplus (\text{Byte } 2) \oplus (\text{Byte } 3) \oplus (0xAA)$$

where ⊕ denotes a bit-wise exclusive OR operation, and 0xAA is the binary byte 10101010.

The 80C32 uses the same formula to calculate the checksum byte contained within the status message sent to the host processor.

7.3 Code Level Interface

In a single microprocessor configuration (see Figure 7-2), the application issues API commands at the code level by directly calling the *_BtControl()* and *_BtStatus()* functions.

7.3.1 _BtControl() Function

The application calls the *_BtControl()* function to issue any of the control commands listed in Section A.1 of Appendix A:.

Use:	`unsigned char _BtControl(unsigned char destination, unsigned char opcode, unsigned char parameter);`
Description:	The destination, opcode, and data parameters correspond to the three command bytes as described in Section 7.1. The opcode field must contain a value corresponding to one of the control commands specified in Section A.1 of Appendix A:.
Effect:	The required command is executed on the bit pump designated by the destination parameter (unless an error condition occurred).
Return Value:	*_PASS (0x00)* indicates a successfully interpreted command. The command is executed. *_FAIL (0x01)* indicates an error condition. The command is not executed. An error condition can be caused by an illegal value in a command field.
Example:	To configure bit pump 2 as a remote terminal (HTU-R), execute this function call: `_BtControl(_BIT_PUMP2, _TERMINAL_TYPE, _HTUR);`

7.3.2 _BtStatus() Function

The application calls the *_BtStatus()* function to issue any of the status request commands listed in Section A.2 of Appendix A:.

Use:

```
unsigned char _BtStatus(unsigned char destination, unsigned
char opcode, unsigned char parameter, char *indication);
```

Description: The destination, opcode, and data parameters correspond to the three command bytes as described in Section 7.1. The opcode field must contain a value corresponding to one of the status request commands specified in Section A.1 of Appendix A:.

Effect: The status parameter is set to the required value (unless an error condition occurred).

Return Value: *_PASS (0x00)* indicates a successfully interpreted command. The value of the status parameter is set correctly.
_FAIL (0x01) indicates an error condition. The value of the status parameter has no meaning and should be ignored. An error condition can be caused by an illegal value in a command field.

Example: To obtain the startup status (e.g., LOS and NMR) of bit pump 2, execute this function call:

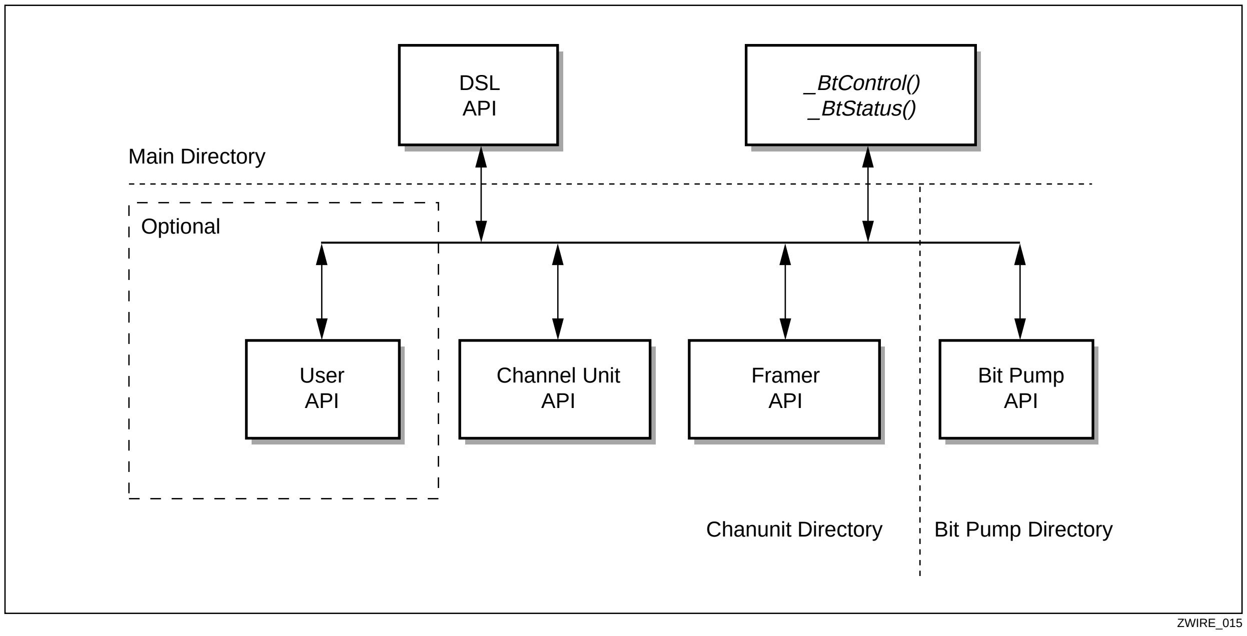
```
_BtStatus((_BIT_PUMP2, _STARTUP_STATUS, 0,
(BP_S_8BIT *) & (bp_status.status));
```

The status is returned in the variable *bp_status*.

7.4 API Hierarchy

ZipWire Version 4.0 added API support for the channel unit and framer devices. The ZipWire software implements the hierarchy illustrated in Figure 7-6. The *_BtStatus()* and *_BtControl()* functions are small functions that parse the destination field in the API command and call the appropriate API function. The non-bit pump API commands illustrated in Figure 7-6 are discussed in Section 7.5.

Figure 7-6. API Hierarchy



7.5 Additional API Commands (User-Defined, DSL, Channel Unit, and Framer)

The code level API structure provides hooks to create user-defined API commands. In a dual processor architecture, the commands can be added without changing the Serial Communication Interface message protocol. Users must only create (define) their own API commands.

_BtStatus() and *_BtControl()* calls the functions listed in Table 7-2 to access the complete set of API commands for the application, bit pump, channel unit, and framer.

Table 7-2. API Functions and File Locations

Function Name	Source File
_BitpumpControl()	bitpump\api.c
_BitpumpStatus()	bitpump\api.c
_CuControl()	chanunit\cu_api.c
_CuStatus()	chanunit\cu_api.c
_DslStatus()	dsl_api.c
_DslControl()	dsl_api.c
_FramerControl()	chanunit\framer_api.c
_FramerStatus()	chanunit\framer_api.c

Because *_BtControl()* and *_BtStatus()* are only parsing functions, the parameters passed and returned in *_BtControl()* and *_BtStatus()* are passed through to the API functions they call. When the ZipWire software is compiled with the CHAN_UNIT directive compiled, Table 7-3 defines the destination field, rather than Table 7-1.

Table 7-3. Destination Field Values

E3	E2	E1	E0	Destination
0	0	0	0	_BIT_PUMP0
0	0	0	1	_BIT_PUMP1
0	0	1	0	_BIT_PUMP2
0	0	1	1	_CU_COMMON
0	1	0	0	_ CU_CHAN1
0	1	0	1	_CU_CHAN2
0	1	1	0	_CU_CHAN3
0	1	1	1	_FRAMER
1	0	0	0	_DSL_APPLICATION
1	0	0	1	_DSL_CHANNEL0
1	0	1	0	_DSL_CHANNEL1
1	0	1	1	_DSL_CHANNEL2

The channel unit and framer destination fields are numbered 3–7 to maintain compatibility with existing EVMs and UIPs. However, these destination fields cause a conflict when the FOUR_LOOPS or SIX_LOOPS compiler flags are specified because those flags specify the destination field between 0–3 and 0–5, respectively. If use of the CHAN_UNIT directive with the FOUR_LOOPS or SIX_LOOPS flags is desired, modify the channel unit destination fields in Table 7-3.

NOTE: The Conexant EVMs are typically compiled with the THREE_LOOPS and CHAN_UNIT compiler flags.

7.6 Opcodes and Parameters

The source file api.h contains C constant definitions for the opcodes of the bit pump API commands and for the possible data field values of commands that use this field.

Table 7-4 and Table 7-5 list the available commands, each with its corresponding C constant opcode and parameter definitions (where applicable). See Appendix A: for details on command use, operation, and parameter setting.

Table 7-4. API Command Summary, Control Commands *(1 of 3)*

Command Name	Opcode C Constants	Parameter Field Options	Parameter Field C Constants
Terminal Type	_TERMINAL_TYPE	HTU-R	_HTUR
		HTU-C	_HTUC
Analog AGC Configuration	_ANALOG_AGC_CONFIG	6-level discrete AGC	_SIX_LEVEL_AGC
Startup Sequence Source	_STARTUP_SEQ_SOURCE	Internal startup sequence	_INTERNAL
		External startup sequence	_EXTERNAL
Regenerator Mode	_REGENERATOR_MODE	Regenerator off	_REGENERATOR_OFF
		Regenerator on	_REGENERATOR_ON
Transmit Scrambler	_TRANSMIT_SCR	Scramble Tx symbols	_ACTIVATE_SCR
		Do not scramble Tx symbols	_BYPASS
Receive Descrambler	_RECEIVE_DESCR	Descramble Rx symbols	_ACTIVATE_DESCR
		Do not descramble Rx symbols	_BYPASS
Data Transfer Format	_FRAMER_FORMAT	Parallel data, clock outputs	_PARALLEL_MASTER
		Parallel data, clock inputs	_PARALLEL_SLAVE
		Serial data	_SERIAL
		Serial data, sign and magnitude swapped	_SERIAL_SWAP
Other Side Conexant	_BT_OTHER_SIDE	Conexant bit pump in other terminal	_BT (Conexant)
		Non-Conexant bit pump in other terminal	_NO_BT (Not Conexant)

Table 7-4. API Command Summary, Control Commands *(2 of 3)*

Command Name	Opcode C Constants	Parameter Field Options	Parameter Field C Constants
LOST Time Period	_LOST_TIME_PERIOD	1-byte unsigned integer	—
Bit Pump On/Off	_SYSTEM_CONFIG	Bit pump on	_PRESENT
		Bit pump off	_NOT_PRESENT
Transmit External Data	_TRANSMIT_EXT_DATA	0	—
Activate	_ACTIVATE	0	—
Deactivate	_DEACTIVATE	0	—
Test Mode	_TEST_MODE	—	_EXIT_TEST_MODE _ANALOG_LOOPBACK _NEAR_LOOPBACK _FAR_LOOPBACK _ISOLATED_PULSE_PLUS3 _ISOLATED_PULSE_PLUS1 _ISOLATED_PULSE_MINUS1 _ISOLATED_PULSE_MINUS3 _FOUR_LEVEL_SCR _TWO_LEVEL_SCR _VCXO_NOMINAL _VCXO_MIN _VCXO_MAX _INTERNAL_ANALOG_ LOOPBACK _ISOLATED_ANALOG_ LOOPBACK _ERLE_TEST _MEASURE_AAGC _ALTERNATING_SYMBOLS_3 _ALTERNATING_SYMBOLS_1
Background and ERLE Test Mode	_ERLE_TEST_MODE	1-byte unsigned integer	—
Symbol Rate	_SYM_RATE _SYM_RATE_HI	10-bit unsigned integer	—
Activation Time-Out	_ACTIVATION _TIMEOUT	30-second time-out	_ACT_TIME_30SEC
		52-second time-out	_ACT_TIME_52SEC
		variable time-out	_ACT_TIME_VARIABLE
Reset	_RESET_SYSTEM	0	—
Enable/Disable	_OPERATE_NLEC	Enable NLEC operation	_NLEC_ON
NonLinear EC (NLEC)		Disable NLEC operation	_NLEC_OFF
Write Transmitter Gain	_WRITE_TX_GAIN	1-byte signed integer	—

Table 7-4. API Command Summary, Control Commands *(3 of 3)*

Command Name	Opcode C Constants	Parameter Field Options	Parameter Field C Constants
Tip/Ring Reversal	_REVERSE_TIP_RING	Tip/Ring Normal	_TIP_RING_NORMAL
		Tip/Ring Reverse	_TIP_RING_REVERSE
BER Meter Start	_BER_METER_START	0	—
BER Meter Stop	_BER_METER_STOP	0	—
Auto Tip/Ring Reversal	_AUTO_TIP_RING	Bypass Auto Tip/Ring	_AUTO_TIP_RING_OFF
		Enable Auto Tip/Ring	_AUTO_TIP_RING_ON
Update ZipStartup Coefficients	_ZIP_START_UPDATE	0	—
ZipStartup Configuration	_ZIP_START_CONFIG	Disable ZipStartup	—
		Enable ZipStartup	—
HCLK Select	_HCLK_SELECT	Default Frequency	_HCLK_DEFAULT
		16 times QCLK	_HCLK_16_TIMES
		32 times QCL	_HCLK_32_TIMES
		64 times QCL	_HCLK_64_TIMES

Table 7-5. API Command Summary, Status Commands (1 of 3)

Command Name	Opcode C Constants	Parameter Field Options	Parameter Field C Constants
Input Signal Level	_SLM	0	—
Input DC Offset	_DC_METER	0	—
Far-End Signal Attenuation	_FELM	0	—
Noise Margin	_NMR	0	—
Timing Recovery Control	_TIMING_RECOVERY _CONTROL	0	—
Bit Pump Status	_STARTUP_STATUS	0	—
Read LEC Coefficient	_LEC_COEFF	Coefficient index	—
Read NLEC Coefficient	_NLEC_COEFF	Coefficient index	—
Read EQ Coefficient	_EQ_COEFF	Coefficient index	—
Read DFE Coefficient	_DFE_COEFF	Coefficient index	—
Software/Chip Version	_VERSION	Major SW Version and Chip Version	_HW_SW_VERSIONS
		Major SW Version	_MAJOR_SW_VERSION
		Minor SW Version	_MINOR_SW_VERSION
		Bit Pump Type and Chip Version	_HW_TYPE_VERSIONS
		ZipStartup Major Version	_ZIP_MAJOR_VERSION
		ZipStartup Minor Version	_ZIP_MINOR_VERSION
Bit Pump Present	_BIT_PUMP_PRESENT	0	—
Self-Test	_SELF_TEST	0	—
Read Bit Pump Register	_REGISTER	Bit pump register address	—

***Table 7-5. API Command Summary, Status Commands** (2 of 3)*

Command Name	Opcode C Constants	Parameter Field Options	Parameter Field C Constants
Bit Pump Configuration	_CONFIGURATION	Read user setup, low byte	_USER_SETUP_LOW_BYTE
		Read user setup, high byte	_USER_SETUP_HIGH_BYTE
		Read LOST time period	_LOST
		Read symbol rate, low value	_BIT_RATE
		Read ERLE configuration byte	_ERLE_SETUP_CONFIG
		Read symbol rate, high value	_BIT_RATE_HI
		Read application setup, low byte	_USER_PARAM_LOW_BYTE
State Number	_STAGE_NUMBER	Read bit pump state number	_ACTIVATION_STAGE
		Read temperature/ environment state number	_TEMP_ENV_STAGE
		Read test mode state number	_TEST_MODE_STAGE
AAGC Value	_AAGC_VALUE	0	—
Read Tx Gain	_READ_TX_GAIN	Request Tx calibration value	_CALIBRATION
		Request Tx gain value	_GAIN
BER Meter Status	_BER_METER_STATUS	BER status	_BER_STATUS
		Bit errors, low byte	_BER_BIT_ERRORS_LOW
		Bit errors, high byte	_BER_BIT_ERRORS_HIGH
		Meter intervals, low byte	_BER_METER_INTERVALS_LOW
		Meter intervals, high byte	_BER_METER_INTERVALS_HIGH

***Table 7-5. API Command Summary, Status Commands** (3 of 3)*

Command Name	Opcode C Constants	Parameter Field Options	Parameter Field C Constants
ERLE Results	_ERLE_RESULTS	SLM low byte	_ERLE_SLM_LOW
		SLM high byte	_ERLE_SLM_HIGH
		FELM low byte	_ERLE_FELM_LOW
		FELM high byte	_ERLE_FELM_HIGH
		SLM2 low byte, bypassing hybrid input	_ERLE_SLM2_LOW
		SLM2 high byte, bypassing hybrid input	_ERLE_SLM2_HIGH
		DC offset low byte	_ERLE_DC_OFFSET_LOW
		DC offset high byte	_ERLE_DC_OFFSET_HIGH
Measure AAGC Results	_AAGC_RESULTS	SLM @ 0 dB, low byte	_AAGC_SLM_0_LOW
		SLM @ 0 dB, high byte	_AAGC_SLM_0_HIGH
		SLM @ 3 dB, low byte	_AAGC_SLM_3_LOW
		SLM @ 3 dB, high byte	_AAGC_SLM_3_HIGH
		SLM @ 6 dB, low byte	_AAGC_SLM_6_LOW
		SLM @ 6 dB, high byte	_AAGC_SLM_6_HIGH
		SLM @ 9 dB, low byte	_AAGC_SLM_9_LOW
		SLM @ 9 dB, high byte	_AAGC_SLM_9_HIGH
		SLM @ 12 dB, low byte	_AAGC_SLM_12_LOW
		SLM @ 12 dB, high byte	_AAGC_SLM_12_HIGH
		SLM @ 15 dB, low byte	_AAGC_SLM_15_LOW
		SLM @ 15 dB, high byte	_AAGC_SLM_15_HIGH

8.0 Diagnostic Tests

This section discusses diagnostic tests that verify correct software and hardware operation, as listed in Table 8-1. Test type "In Service" means that the DSL connection and the data stream are not effected by the test.

Table 8-1. Overview of Diagnostic Tests

Test	Purpose	Test Type	Comments
Oscilloscope Probing of Tx Lines	Checks if the bit pump is transmitting	In Service	Easy test to set up, provides quick visual data.
TDEBUG	Diagnoses startup problems	In Service	Compile code with #TDEBUG directive.
Test Modes	Checks transmit power and runs various loopback tests	Out of Service	Requires software modifications or UIP program.
ERLE	Verifies the hybrid circuit	Out of Service	Critical test. Requires software modifications or UIP program.
BER Meter	Measures the Bit Error Rate (BER)	Out of Service	Requires software modifications or UIP program.

8.1 Oscilloscope Probing of Tx Lines

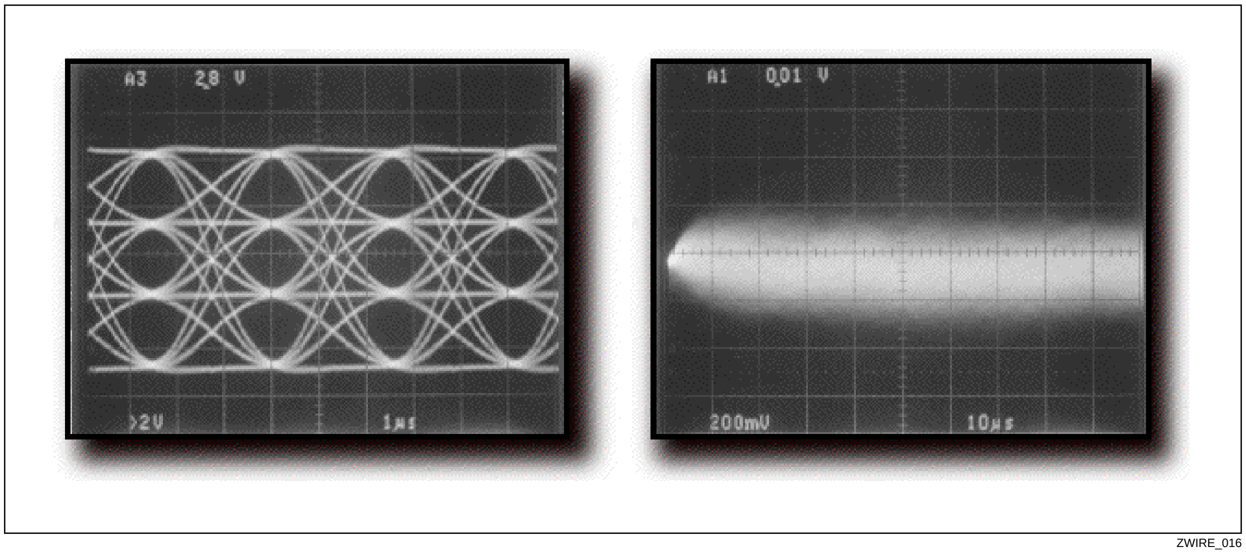
To examine the bit pump's transmitted output, follow these steps:

1. Connect an oscilloscope probe to the circuit side (the side connected to the output of the bit pump) of the line driver transformer.
2. Trigger the oscilloscope using the QCLK signal from the bit pump on the HTU-C.

NOTE: The QCLK output of the HTU-R drifts until it phase locks on the 2-level signal initially transmitted by the HTU-C.

When the bit pump is transmitting a 2-level, all 1s signal, the expected wave form is a sinusoidal signal with a frequency of one-quarter of the data rate (data rate / 4) and amplitudes of approximately ±2.64 V. When the bit pump is transmitting a 4-level, all 1s signal, the expected wave form is a sinusoidal signal with a frequency of one-quarter the data rate and amplitudes of ±2.64 V and ±0.88 V. A typical four-level wave form is called an eye-pattern, as illustrated on the left side of Figure 8-1. The left side shows the signal transmitted from one bit pump; the right side shows the distorted signal received by the bit pump on the far end. The bit pump on the far end filters the distorted signal and recovers the signal and data using its Digital Signal Processing (DSP) algorithms.

Figure 8-1. Four-Level Eye Pattern



8.2 DEBUG

As mentioned in Section 5.7, the TDEBUG directive enables the compilation of *printf()* statements within the ZipWire software. This option provides an output that can be sent to Conexant for examination. Examples of TDEBUG outputs are given in Figure 8-2 for an HTU-C and Figure 8-3 for an HTU-R, both running at a 1,552 Kbps data rate and a zero-length DSL. Comments have been added to the HTU-R output to provide more explanation. The output varies based on the DSL rate, DSL conditions, and DSL length. Hardware problems, such as a suboptimum hybrids, cause various startup problems, such as signal not detected, lack of phase locking by HTU-R, etc.

Figure 8-2. Example of TDEBUG Output from HTU-C *(1 of 2)*

```
Bitpump present
2: Bitpump not present
3: Bitpump not present
1: Reset system
1: Load MicroCode
1: Init regs
Bt8973 detected.
1: Bitpump SW Version 4.3
1: Reset system
1: Load MicroCode
1: Init regs
HTU-C #1: Data Rate = 1168 Kbps
HTU-C #1: State = Configuration State
1: Reset system
1: Load MicroCode
1: Init regs
HTU-C #1: Data Rate = 1168 Kbps
HTU-C #1: Activating...
HTU-C #1: Cold Startup Attempt...
HTU-C #1: State = Activating State
1: Activate HTU-C
1: Init regs
1: DC Offset -54
1: TX CS0
1: Adapt EC
1: felm = 496
1: 0 length detected
Adjust FELM High TH = 1008
1: Wait for signal...
 LOS OFF
1: Signal detected
1: FELM = 16256
FFE Init Scale=3 for High Data Rates
1: SLM @ 15dB = 23424
1: SLM @ 12dB = 21120
1: SLM @ 9dB = 17216
1: SLM @ 6dB = 12480
1: SLM @ 3dB = 8640
1: SLM @ 0dB = 6208
Adapt EC after AAGC
1: SLM after AAGC = 6400
1: FELM after AAGC = 6272
current gain = 0 dB
1: FELM = 6208
1: Search opt phase
(0) F = 2166, D = 320 P=177 NLM=192
(1) F = 1671, D = 336 P=172 NLM=304
(2) F = 2549, D = 352 P=253 NLM=272
(3) F = 3070, D = 368 P=1 NLM=192
(4) F = 1371, D = 336 P=0 NLM=176
(5) F = 2908, D = 352 P=85 NLM=160
(6) F = 2849, D = 352 P=124 NLM=192
(7) F = 3087, D = 304 P=171 NLM=160
(8) F = 2914, D = 304 P=216 NLM=176
(9) F = 2721, D = 320 P=233 NLM=160
(10) F = 2929, D = 288 P=250 NLM=160
(11) F = 2973, D = 272 P=263 NLM=144
```

Figure 8-2. Example of TDEBUG Output from HTU-C *(2 of 2)*

```
(12) F = 2885, D = 272 P=265 NLM=144
(13) F = 2557, D = 304 P=240 NLM=160
(14) F = 2809, D = 272 P=223 NLM=160
(15) F = 2527, D = 288 P=210 NLM=192
Number of Opened Phases = 15
Selected as BAD Phase Quality: 4(0) 3(1) 5(85) 0(1) 1(1) 2(1) 6(1) 7(1) 8(1)
1: Selected as High Noise Phases: 1(304) 2(272) 0(192) 3(192) 6(192)
Phases = [1 1 1 1 0 0 1 1 1 233 250 263 265 240 223 210 ]
1: Optimal phase = 12
1: Adapt EC @ opt phase
ReOpen Eye At Optimal Phase Completed
1: DAGC (After Reopen Eye) = 288
1: FFE = [
-14 ;46 ;-136 ;410 ;-1079 ;2878 ;326 ;-370 ;]
1: 1: DAGC = 288
1: FFE = [
-14 ;46 ;-136 ;410 ;-1079 ;2878 ;326 ;-370 ;]
1: TX CS1
1: Wait for 4level...
HTU-C #1: State = Activating State S1
HTU-C #1: State = Active Rx State
1: 1: DAGC = 288
1: FFE = [
-14 ;46 ;-136 ;410 ;-1079 ;2878 ;326 ;-370 ;]
1: 4-level detected
1: BER Meter = 0
1: DAGC Target = 1071
1: 1: DAGC = 288
1: FFE = [
-14 ;46 ;-136 ;410 ;-1079 ;2878 ;326 ;-370 ;]
1: DFE =[
 -3787 ;-3320 ;-1392 ;-1277 ;-1117 ;-1015 ;-895 ;-782 ;];
1: Adapt NLEC & EP
Non normlized FELM: 2434
1: Final 1: DAGC = 304
1: FFE = [
-25 ;44 ;-148 ;359 ;-1022 ;2653 ;300 ;-357 ;]
1: Normal operation
HTU-C #1: Startup Time = ~13 sec
HTU-C #1: State = Active Tx/Rx State
```

Figure 8-3. Example of TDEBUG Output from HTU-R *(1 of 3)*

```
Bitpump present
2: Bitpump not present
3: Bitpump not present
1: Reset system
1: Load MicroCode
1: Init regs
Bt8973 detected.
1: Bitpump SW Version 4.3
1: Reset system
1: Load MicroCode
1: Init regs
HTU-R #1: Data Rate = 1168 Kbps
HTU-R #1: State = Configuration State
1: Reset system
1: Load MicroCode
1: Init regs
HTU-R #1: Data Rate = 1168 Kbps
HTU-R #1: Activating...
HTU-R #1: Cold Startup Attempt...
HTU-R #1: State = Activating State
1: Activate HTU-R
1: Init regs
1: DC Offset 78
1: Wait for signal...
 LOS OFF
 LOS ON
 LOS OFF
1: Signal detected
1: felm = 16832
1: Open eye
1: Freq cell 0
1: FELM = 15872
```

Comment: Detected large FELM, which is normal for short DSL lengths. At max reach (no noise) lengths, the expected FELM ranges from 300–500.

Figure 8-3. Example of TDEBUG Output from HTU-R *(2 of 3)*

```
FFE Init Scale=3 for High Data Rates
Freq Cell: 0, Attempts: 0
1: Freq Acqu
1: Verify phase locking
VCXO = –1657
1: PDM(0) = 0 ; PDM(1) = 3 ; PDM(2) = 1 ; PDM(3) = –1 ;
1: Avg PDM = 0 ; |Avg PDM| = 2
 1: Search opt phase
(0) F = 2758, D = 224 P=346 NLM=80
(1) F = 2609, D = 240 P=333 NLM=96
(2) F = 2861, D = 208 P=332 NLM=112
(3) F = 2693, D = 208 P=327 NLM=80
(4) F = 2620, D = 208 P=298 NLM=96
(5) F = 2300, D = 240 P=249 NLM=160
(6) F = 2204, D = 224 P=228 NLM=208
(7) F = 2199, D = 240 P=411 NLM=320
(8) F = 2914, D = 240 P=157 NLM=240
(9) F = 3493, D = 240 P=2 NLM=80
(10) F = 892, D = 240 P=0 NLM=336
(11) F = 2829, D = 272 P=126 NLM=80
(12) F = 3134, D = 240 P=178 NLM=112
(13) F = 2826, D = 256 P=230 NLM=80
(14) F = 2885, D = 240 P=274 NLM=80
(15) F = 3428, D = 192 P=307 NLM=80
```

Comment: The 16-phase quality values (indicated after “P=”) should have a Gaussian shape that corresponds to the shape of an eye pattern. The maximum P (minus some high NLM (low SNR) readings) should produce the optimal phase. In this case, Phase 0 has P = 346.

Note: If the phase search yield less than 5 good phases (i.e., P > 0), the software reattempts the phase search with a different set of FFE coefficients. In this case, there were 15 good phases.

```
Selected as BAD Phase Quality: 10(0) 9(2) 11(126) 6(1) 7(1) 8(1) 12(1) 13(1) 14(
1)
1: Selected as High Noise Phases: 10(336) 7(320) 8(240) 6(208) 5(160)
Phases = [346 333 332 327 298 1 1 1 1 0 1 0 1 1 1 307 ]
1: Optimal phase = 0
ReOpen Eye
DAGC Target = 1374
1: TX RS0
1: SLM @ 15dB = 28288
1: SLM @ 12dB = 27008
1: SLM @ 9dB = 22016
1: SLM @ 6dB = 15680
1: SLM @ 3dB = 11072
1: SLM @ 0dB = 7872
```

Comment: The software reduces the AAGC gain until the SLM < 5,300 or the gain is reduced to 0 dB. The software selected the 0 dB setting, which is correct for short DSL lengths.

Figure 8-3. Example of TDEBUG Output from HTU-R *(3 of 3)*

```
FFE Coefficients (Before Scaling):
1: FFE = [
–16 ;1 ;–71 ;206 ;–835 ;3028 ;46 ;–111 ;]
1: DAGC (Before Scaling) = 208
1: DAGC (Before 2nd Scaling) = 208
1: Adapt EC
1: Before Self: 1: DAGC = 208
1: After Self: 1: DAGC = 336
1: After Readapt: 1: DAGC = 304
1: Restart EQ: 1: DAGC = 304
1: Waiting for 4 level...
```

Comment: The code adapted the equalizers (DAGC and FFE) and is waiting for 4-level. The value of DAGC is ~400, which is good for short DSL lengths; the valid range is 0–2,032.

```
1: 4 level detected
1: BER Meter = 0
1: TX RS1
1: FFE = [
–16 ;1 ;–71 ;206 ;–835 ;3028 ;46 ;–111 ;]
1: DFE =[
 –2681 ;–2669 ;–1721 ;–1528 ;–1343 ;–1170 ;–1030 ;–912 ;];
HTU-R #1: State = Activating State S1
1: Adapt NL EC & EP
HTU-R #1: State = Active Rx State
Non normlized FELM: 2277
1: Normal operation
HTU-R #1: Startup Time = ~13 sec
HTU-R #1: State = Active Tx/Rx State
```

8.3 Test Modes

The bit pump can be operated in special test modes—used during development and field operation—for maintenance and fault identification. All test modes are activated using the Test Mode command, with the command parameter value selecting the specific mode.

To exit a test mode, issue the Test Mode command with the _EXIT_TEST_MODE (0x00 value) parameter. The action taken when exiting a test mode condition depends on the specific test mode selected. Activating most test modes while the bit pump is synchronized causes the bit pump to lose sync. Exiting the test mode and returning to normal data transfer condition requires a full restart. In these cases, the bit pump ASM is set to the Idle state when the test mode is exited.

Other test modes do not create a sync loss condition, and normal operation can continue when exiting the test condition. When exiting one of these test modes, the bit pump returns to the state that was active when the Test Mode command was issued.

Table 8-2 describes the available test modes, and the Sync Lost column indicates if synchronization is lost when the test mode is exited. Further information can be found in the Test Mode Command description in Appendix A:.

Table 8-2. Bit Pump Test Modes *(1 of 2)*

Test Mode	Description	Sync Lost When Exited
External analog loopback (hybrid inputs disabled)	Transmits the externally-supplied Tx symbols, uses the echo signal as a received signal, and detects the symbols using the standard equalizer.	Yes
Digital near loopback	Tx symbols supplied to the bit pump are looped back to the Rx symbols from the bit pump.	No
Digital far loopback	Detected (Rx) symbols are transmitted back on the loop.	No
Transmit isolated +3 pulse	Transmits (repeatedly) an isolated +3 level pulse. Useful for testing the transmitted pulse shape.	Yes
Transmit isolated +1 pulse	Transmits (repeatedly) an isolated +1 level pulse.	Yes
Transmit isolated –1 pulse	Transmits (repeatedly) an isolated –1 level pulse.	Yes
Transmit isolated –3 pulse	Transmits (repeatedly) an isolated –3 level pulse.	Yes
Continuous 4-level transmission	Continuous transmission of a 4-level scrambled 1s sequence (internally generated). Useful for measuring transmitted power and spectral shape.	Yes
Continuous 2-level transmission	Continuous transmission of a 2-level scrambled 1s sequence (internally generated).	Yes
Set minimum VCXO frequency	Sets VCXO control word to its minimum value.	Yes

Table 8-2. Bit Pump Test Modes *(2 of 2)*

Test Mode	Description	Sync Lost When Exited
Set nominal VCXO frequency	Sets VCXO control word to its nominal value. Useful for measuring VCXO center frequency.	Yes
Set maximum VCXO frequency	Sets VCXO control word to its maximum value.	Yes
Internal analog loopback (transmitting loopback)	Transmits the externally supplied Tx symbols out the TXP and TXN pins and detects the symbols on the hybrid inputs (RXBP and RXBN); the receive inputs (RXP and RXN) are bypassed.	Yes
Isolated analog loopback (silent loopback)	The externally supplied Tx symbols are internally looped back in the bit pump. The transmitter (TXP and TXN) is turned off (silent).	Yes

8.3.1 Loopback Test Modes Overview

The digital and analog loopback modes are illustrated in Figure 8-4. The digital near loopback is useful for testing the bit pump interface to the network.   The analog loopbacks perform a similar function but test additional hardware blocks within the signal path. The digital far loopback is useful for testing full 2-way transmission over the DSL. This loopback requires that the bit pump internal Tx scrambler and Rx descrambler be enabled on the far-end bit pump. Table 8-3 lists the loopback tests and those that require a DSL activation. The mini-activation required for the analog loopbacks is implemented in the *_HandleTestMode()* state machine.

Figure 8-4. Loopback Modes

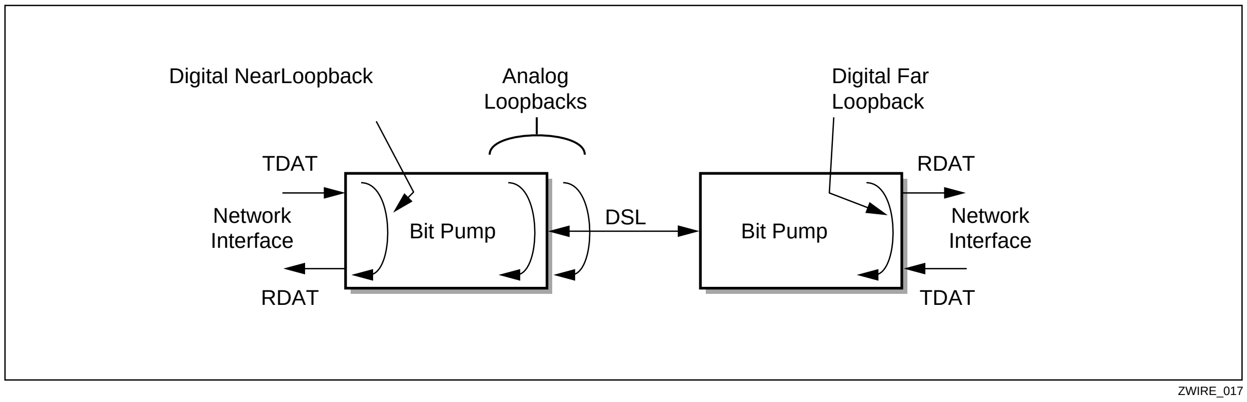


Table 8-3. DSL Activation Requirements for Loopback Modes

Test	DSL Activation Required
Digital near loopback	No
Analog loopbacks	Local mini-activation (DSL connection to the other side is not required)
Digital far loopback	Yes

8.3.2 Analog Loopback Test Modes

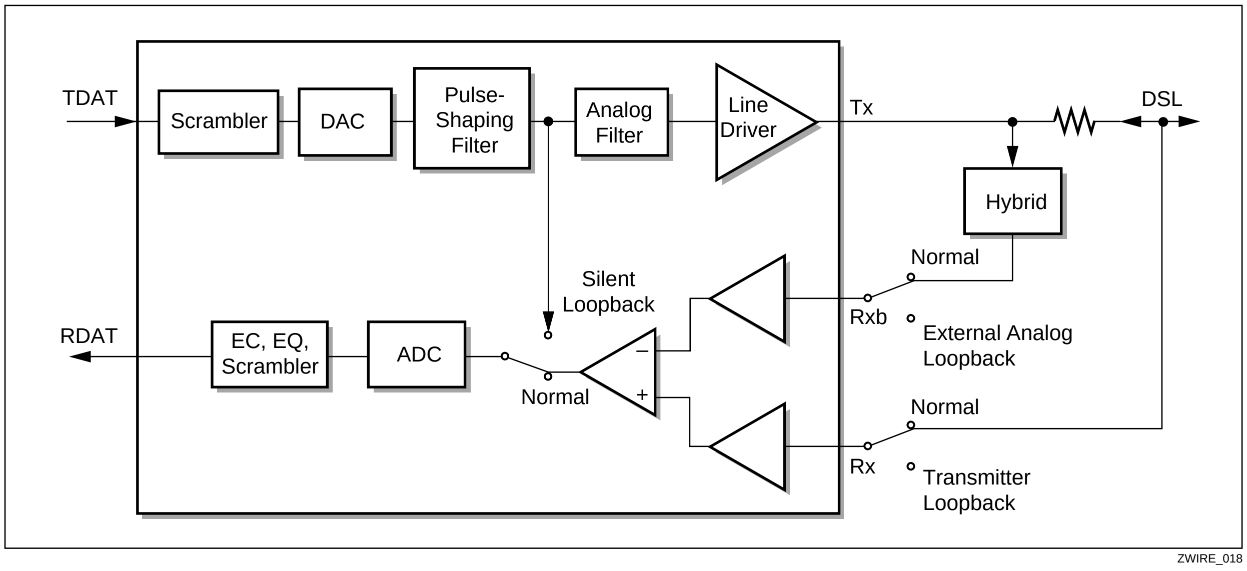
The set of analog test modes are as follows:

- Transmitter loopback or internal analog loopback (recommended)
- External analog loopback
- Silent loopback or isolated analog loopback

The three analog loopback test modes are illustrated in Figure 8-5. These tests loop the TDAT data path back to the RDAT data path. These tests are set up by configuring the ADC control register (0x21) in the bit pump. If the other side is not transmitting a signal, the three analog loopback tests provide nearly the same RDAT signal.

As Figure 8-5 illustrates, the analog loopbacks return the TDAT signal at different signal points. The transmitter loopback is recommended because it tests the performance of the hybrid circuit.

Figure 8-5. Analog Loopback Test Modes



8.4 Internal BER Meter Operation

This section describes how to use the bit pump’s internal Bit Error Rate (BER) meter. The BER meter code is only accessible when the BER_METER directive is defined. The BER meter can be used to verify the integrity of the line. It can also be used as a diagnostic tool during production and field testing or hardware and software development.

8.4.1 Creating BER Meter Test Code

The BER meter uses the bit pump's internal scrambled 1s generator and descrambler to detect bit errors. For the BER meter to function properly, both the HTU-C and HTU-R must issue the _BER_METER_START API command. Because the BER meter replaces the entire DSL stream with scrambled 1s, it cannot be used when transporting real payload data. The BER meter is only operational after the bit pumps on the HTU-C and HTU-R have successfully completed startup. All accesses to the BER meter are performed using API commands (see Appendix A:).

The BER meter test can be run either by modifying the ZipWire software in a single processor configuration or by using the UIP program or API commands in a dual processor configuration. The necessary directives to run in these configurations are listed in Table 8-4.

Table 8-4. Required Directives for Creating BER Meter Executable

Configuration	Directives	Software Setup
Single Processor	C51, ADD_DELAY, PDATA_MODE, HTU-C or HTU-R, BER_METER, TDEBUG	See Section 8.4.2 for software modifications
Dual Processor (using API commands)	C51, ADD_DELAY, PDATA_MODE, HTU-C or HTU-R, BER_METER, SER_COM	Use the _BER_METER_START, _BER_METER_STATUS, and _BER_METER_STOP API commands
Conexant EVM and UIP	C51, ADD_DELAY, PDATA_MODE, HTU-C or HTU-R, BER_METER, SER_COM	Use the BER meter test command on the Startup Form screen

8.4.2 Software Modifications for a Single Processor Architecture

When the application ASM reaches ACTIVE_TX_RX_STATE, the application code must call the _BER_METER_START API command to start the BER meter test. Because the BER_METER_START API commands must be issued on both the HTU-R and HTU-C, there is a chance that one side may initially report false BER errors. When starting the test, the _BER_METER_START API command can be called more than once to zero out the *bit_errors* counter.

After the test is started, the BER_METER_STATUS API command and *printf()* statements can provide the test results. The *get_ber_meter_status()* function provides example code to output the BER results:

```
/*
 * Assuming 288 Kbps Data Rate, Normal Operation, and BER Meter Active
 * Also assumes using compiler/linker that supports floating point.
 */
void get_ber_meter_status (unsigned char no)
{
   unsigned char temp, temp1;
   unsigned int errors, intervals;
   float avg_ber, elapsed_time;

   _BtStatus(no, _BER_METER_STATUS, _BER_BIT_ERRORS_LOW, &temp);
   _BtStatus(no, _BER_METER_STATUS, _BER_BIT_ERRORS_HIGH, &temp1);
   errors = (unsigned)BYTE2WORD(temp1, temp);

   _BtStatus(no, _BER_METER_STATUS, _BER_METER_INTERVALS_LOW, &temp);
   _BtStatus(no, _BER_METER_STATUS, _BER_METER_INTERVALS_HIGH, &temp1);
   intervals = (unsigned)BYTE2WORD(temp1, temp);

   /* equations don’t show necessary type casting */
   avg_ber = (errors) / (intervals * 0x8000 * 2);
   elapsed_time = (intervals * 0x8000 * 2) / (288000);

#if TDEBUG
   printf(“# Bit Errors = %u\n”, errors);
   printf(“# Meter Intervals = %u\n”, intervals);
   printf(“Avg Ber = %.2e\n”, avg_ber);
   printf(“Elapsed Time = %.1f seconds\n”, elapsed_time);
#endif

   return;
}
```

The *bit_errors* variable is a 16-bit unsigned integer that holds up to 65,535 errors. The *meter_interval* variable is a 16-bit unsigned integer that holds up to 65,536 meter intervals. At 1,168 Kbps, a meter interval of 32,768 symbols takes 56 ms. Therefore, *meter_interval* can count the number of meter intervals that occur in (65,536 x 56) ms, i.e., approximately 1 hour.

The interrupt handler reads the bit error rate meter register (0x4C, 0x4D) and updates the *bit_errors* and *meter_intervals* counters after every meter interval. This causes the interrupt handler to be slightly longer, ~55 µs per bit pump on an 80C32 processor running at 11.0592 MHz.

8.4.3 Calculation of BER Meter Results

Use these equations to compute the average BER and elapsed test time:

$$\text{Avg BER} = \frac{\text{Number of Bit Errors}}{\text{Number of Meter Intervals} \times \text{Meter Interval Length} \times 2}$$

$$\text{Elapsed Time} = \frac{\text{Number of Meter Intervals} \times \text{Meter Interval Length} \times 2}{\text{Data Rate}}$$

The variables for these equations are explained in Table 8-5.

Table 8-5. Calculation of BER Meter Results

Variable	How Derived
Number of Bit Errors	Read the Number of Bit Errors, Low and High Byte API commands and build a 16-bit unsigned integer.
Number of Meter Intervals	Read the Number of Meter Intervals, Low and High Byte API commands and build a 16-bit unsigned integer.
Meter Interval Length	Read the bit pump Meter Interval register (address 0x18, 0x19) and build a 16-bit unsigned integer. During normal operation, these registers should always read 0x8000 (32,768 symbols).
Data Rate	Data rate of the system, e.g., 288,000 Kbps.
×2	The ×2 is necessary because there are 2 bits per symbol and the meter interval length is based on the number of symbols.
NOTE(S): 16-bit value = (high byte << 8) + (low byte)	

8.5 ERLE Diagnostic Code

The Echo Return Loss Enhancement (ERLE) program is a diagnostic tool used to verify the integrity of the hardware’s analog front-end and echo canceler. The ERLE program is useful when modifying the hybrid for different data rates.

8.5.1 Creating ERLE Test Code

The ERLE test can be run either by modifying the ZipWire software in a single processor configuration or by using the UIP program or API commands in a dual processor configuration. Table 8-6 lists the directives needed to run in these configurations.

The ERLE test runs the same software on the HTU-C and HTU-R. The HTU-C or HTU-R directives, or both, must be defined when compiling the software.

Table 8-6. Required Directives for Creating ERLE Executable

Configuration	Directives	Software Set Up
Single Processor	C51, ADD_DELAY, PDATA_MODE, HTU-C or HTU-R, ERLE, TDEBUG	See Section 8.5.2 for software modifications
Dual Processor (using API commands)	C51, ADD_DELAY, PDATA_MODE, HTU-C or HTU-R, ERLE, SER_COM	Use the _ERLE_TEST_MODE and _ERLE_RESULTS API commands
Conexant EVM and UIP	C51, ADD_DELAY, PDATA_MODE, HTU-C or HTU-R, ERLE, SER_COM	Use the ERLE test mode commands on the Test and Utility screen

8.5.2 Software Modifications for a Single Processor Architecture

The only modification needed is to add one line of code that sets the configuration and activates the test. To run the test at 768 Kbps, the required code is

```
dip_sw.port1 = 0x21;
```

The following code identifies where to place the additional line in the *main()* function in *dsl_main.c*:

```
#if defined (HTUC) && !defined (HTUR)
        dip_sw.bits.terminal_type = _HTUC;
#endif /* HTUC only */

#if defined (HTUR) && !defined (HTUC)
        dip_sw.bits.terminal_type = _HTUR;
#endif /* HTUR only */

       /* Setup the dip_sw.port1 here to run ERLE.
        * Run the ERLE test at 1,168 Kbps and AAGC = 12 dBm.
        */
        dip_sw.port1 = 0x21;
        /*
         * Configure EVMs, must perform in this order
         *      - Enable Bit Pump
         *      - Init/Config CU EVM
         *      - Configure Bit Pump
         *      - Handle Test Modes
         */
        for (bp = 0; bp < _NO_OF_LOOPS; bp++)
            {
            _EnableBitpump(bp);
            }
#ifdef CHAN_UNIT
        _InitChannelUnitEvmBoard();
#endif
        for (bp = 0; bp < _NO_OF_LOOPS; bp++)
            {
            _ConfigureBitpump(bp);
            _HandleTestModes(bp);
            .
            .
            .
```

If the *main()* function has been modified, follow these guidelines on where to set the dip_sw.port1 byte:

- Set this byte before calling *_ConfigureBitpump(bp)* and *_HandleTestModes()*
- Set this byte after any other code modifies bits in the *dip_sw* byte or the entire byte

To run the ERLE test at other data rates, change the value of the *dip_sw.port1* variable. Appendix D: provides background information on the *dip_sw.port1* structure.

8.5.3 Running the ERLE Test, No DSL

The following items are required to run the ERLE test:

- ERLE software
- 150 Ω resistor
- RS232 cable
- Terminal emulator (i.e., QMODEM, KERMIT, Hyper Terminal, or equivalent)

The following four steps outline the process for running ERLE:

1. Download the ERLE software or install an EPROM containing the ERLE software.
2. Connect the 150 Ω resistor to the DSL side of the transformer. The resistor provides an impedance mismatch that in turn creates an echo.
3. Connect the RS232 cable from the EVM to the computer and run a terminal emulator (9,600 baud, 8 bits, no parity, 1 stop bit).
4. Power on—Reset the system and run the program. The results will scroll on the terminal emulator.

8.5.4 Running the ERLE Test, Simulated or Real DSL

Using a simulated or real DSL provides more accurate test results because the returned echo is more representative of a real system. The DSL is not active; it only provides a passive resistance. The following items are required to run the ERLE test using a DSL:

- ERLE software
- a DSL simulator and a passive termination (either a 135 Ω resistor or a bit pump device)
- RS232 cable
- terminal emulator (i.e., QMODEM, KERMIT, Hyper Terminal, or equivalent)

The following four steps outline the process for running ERLE:

1. Download the ERLE software or install an EPROM containing the ERLE software.
2. Connect a second (unpowered or powered down with the transmitter off) bit pump line card or a 135 Ω resistor and a 12,400 foot (26 AWG) simulated DSL to the DSL side of the transformer.
3. Connect the RS232 cable from the EVM to the computer and run a terminal emulator (9,600 baud, 8 bits, no parity, 1 stop bit).
4. Power on—Reset the system and run the program. The results will scroll on the terminal emulator.

8.5.5 Typical ERLE Output, No DSL

The output from the HTU-R and HTU-C is nearly the same if the hybrid circuit is the same design. An output from the HTU-R is illustrated in Figure 8-6

Figure 8-6. Typical ERLE Output, No DSL

```
Bit Pump present
2: Bit pump not present
3: Bit pump not present
1: Reset system
1: Load MicroCode
1: Init regs
Bt8973 detected.
1: Bit pump SW Version 4.2
1: Reset system
1: Load MicroCode
1: Init regs
HTU-C #1: Data Rate = 1168 Kbps

Background Test—Transmitter OFF
1: Test Mode
1: ERLE
1: Init regs
1: AAGC 12 dB
1: DC Offset = 5
1: EC Highest, Higher, High 1: Bypass NLEC.
1:Measure ...
HTU-C #1: SLM = 4 FELM = 4

ERLE Test—Transmitter ON, 4 Level
1: Test Mode
1: ERLE
1: Init regs
1: AAGC 12 dB
1: DC Offset = 1
1: EC Highest, Higher, High 1: Bypass NLEC
1: Measure ...
HTU-C #1: SLM = 3399 FELM = 14 SLM/FELM = 242 ERLE = 47.70
HTU-C #1: BYPASS HYBRID: SLM2 = 19299 SLM2/SLM = 5
ANALOG ERLE = 15.08

AAGC Check—Transmitter ON, 4 Level
1: Test Mode
1: Measure AAGC
1: Init regs
HTU-C #1: AAGC 0 dB: SLM = 851 Gain = 0.00
HTU-C #1: AAGC 3 dB: SLM = 1201 Gain = 2.99
HTU-C #1: AAGC 6 dB: SLM = 1687 Gain = 5.94
HTU-C #1: AAGC 12 dB: SLM = 3382 Gain = 11.98
HTU-C #1: AAGC 15 dB: SLM = 4796 Gain = 15.02

Test Complete...
```

8.5.6 Expected Results

Table 8-7 lists the expected results without a DSL connection.

Table 8-7. ERLE Results (No DSL, AAGC = 12 dBm, 1,168 Kbps, 150 Ω Termination)

Result	Min	Typical	Max
DC Cancel	–200	50	200
Background SLM	0	2	5
Background FELM	0	2	5
Transmit ON SLM	3,000	3,400	3,800
Transmit ON FELM	0	9	15
Transmit ON ERLE	52	57	62
Transmit ON AERLE	7	10	15
AAGC 0 dB	0	0	0
AAGC 3 dB	2.8	3.0	3.2
AAGC 6 dB	5.8	6.0	6.2
AAGC 9 dB	8.8	9.0	9.2
AAGC 12 dB	11.8	12.0	12.2
AAGC 15 dB	14.8	15.0	15.2

Table 8-8 list the expected results with a bit pump termination at 12,400 feet.

Table 8-8. ERLE Results (DSL, AAGC = 12 dBm, 1,168 Kbps, 12,400 Feet (26 AWG), Bit Pump Termination)

Result	Min	Typical	Max
DC Cancel	–200	0	200
Background SLM	0	1	4
Background FELM	0	1	4
Transmit ON SLM	2,400	2,800	3,200
Transmit ON FELM	0	2	5
Transmit ON ERLE	56	63	70
Transmit ON AERLE	15.5	17.5	19.2
AAGC 0 dB	0	0	0
AAGC 3 dB	2.7	3.0	3.3
AAGC 6 dB	5.7	6.0	6.3
AAGC 9 dB	8.7	9.0	9.3
AAGC 12 dB	11.7	12.0	12.3
AAGC 15 dB	14.7	15.0	15.3

8.5.7 Interpretation of Results

There are three major portions to the ERLE test (see Figure 8-6):

Test 1. Background Noise Test—Transmitter OFF
Test 2. ERLE Test—Transmitter ON
Test 3. AAGC Check

The ERLE test measures the performance of the analog (external hybrid) and digital (internal to bit pump) echo cancellation functions. The values shown in this section should not be taken as expected values; they are used only for examples.

Test 1. Background Noise Test—Transmitter OFF

The background test measures the Signal Level Meter (SLM) while the transmitter is turned off. This effectively gives the amount of noise present at the input of the A/D.

```
Background Test—Transmitter OFF
1: Test Mode
1: ERLE
1: Init regs
1: AAGC 12 dB
1: DC Offset = 5
1: EC Highest, Higher, High
1: Bypass NLEC
1:Measure ...
HTU-C #1: SLM = 4 FELM = 4
```

The SLM and Far-End Level Meter (FELM) provide two measurements of the received signal strength after echo cancellation. The SLM reading is measured after analog echo cancellation only; the FELM reading is measured after both analog and digital echo cancellations. See the *RS8973 Single-Chip SDSL/HDSL Transceiver Data Sheet* (100nnnx, formerly N8973DSD) for the location of these measurements with the bit pump.

The SLM and FELM values are important in Test 1. Because the transmitter is not turned on, these values should both be very small; the maximum expected values are 5 for both. The SLM and FELM readings are 16-bit unsigned numbers. The DC offset is typically somewhere between –100 to +100.

A large SLM and FELM with the transmitter off usually indicates a DSL connection to a remote (that is transmitting).s

Test 2. ERLE Test—Transmitter ON

The ERLE result determines the amount of echo canceled. The echo is canceled initially by the external hybrid and then by the Linear Echo Canceler (LEC) in the bit pump. The Analog ERLE (AERLE) determines how much echo is canceled by the external hybrid.

```
ERLE Test—Transmitter ON, 4 Level
1: Test Mode
1: ERLE
1: Init regs
1: AAGC 12 dB
1: DC Offset = 1
1: EC Highest, Higher, High
1: Bypass NLEC
1: Measure ...
HTU-C #1: SLM = 3399 FELM = 14 SLM/FELM = 242 ERLE = 47.70
HTU-C #1: BYPASS HYBRID: SLM2 = 19299 SLM2/SLM = 5
ANALOG ERLE = 15.08
```

The SLM, FELM, and SLM2 measurements are important. The SLM measures the input to the A/D converter. The FELM measures the output of the Digital Echo Canceler (DEC). The FELM value should be small (typically less than 9) because there is no far-end transmitter. The SLM/FELM ratio gives a quality measurement of the digital echo cancellation; the larger the ratio, the better the performance of the digital echo cancellation.

NOTE: A large SLM or FELM with the transmitter on could indicate that the DSL is not properly terminated.

The SLM2 measures the input to the A/D converter when bypassing the analog hybrid. The SLM2/SLM ratio gives a quality measurement of the analog echo cancellation.

The ERLE and AERLE are dB calculations defined as

$$\text{ERLE} = 20 * \log (\text{SLM} / \text{FELM})$$
$$\text{AERLE} = 20 * \log (\text{SLM2} / \text{SLM})$$

The expected value of FELM is ~9 and the expected value of ERLE is ~57. The sum of ERLE and AERLE should be ~66.

If the ERLE test is run with different AAGC settings, the SLM and SLM2 measurements change. The FELM and ERLE may vary if a 1 µF DC-blocking capacitor is used. The results are also worse if line protection circuitry is part of the system.

A high FELM could be due to incorrect component values, component tolerances other than the recommended resistors = 1%, and capacitors = 5%. Capacitors that are not of Negative-Positive Zero (NP0) type, as recommended by Conexant, could cause a problem. A transformer that is not approved by Conexant could also be a contributing factor. The ERLE test indicates possible areas to debug, but it does not point to the exact resistor, capacitor, or other component that could be causing the problem.

Test 3. Analog Automatic Gain Control (AAGC) Check

The AAGC Check ensures that the gain settings are monatomic.

```
AAGC Check—Transmitter On, 4-Level
1: Test Mode
1: Measure AAGC
1: Init regs
HTU-C #1: AAGC 0 dB: SLM = 851 Gain = 0.00
HTU-C #1: AAGC 3 dB: SLM = 1201 Gain = 2.99
HTU-C #1: AAGC 6 dB: SLM = 1687 Gain = 5.94
HTU-C #1: AAGC 12 dB: SLM = 3382 Gain = 11.98
HTU-C #1: AAGC 15 dB: SLM = 4796 Gain = 15.02
```

Appendix A: API Command Set Reference

This appendix contains a reference guide to the ZipWire API command set. All available commands are listed, giving information on command use, application, syntax, and options.

The set of commands is divided into three groups:

1. Control commands—Control the operation modes of the bit pump and set various parameters.
2. Status request commands—Inquire for status and monitoring information from the bit pump. A 1-byte response containing the required information is returned.
3. The Acknowledge Message.

A.1 Control Commands

For each command, a description of its operation, opcode, and options is given. Commands that select bit pump operational options and set bit pump parameters have their default values identified. The default values are also listed in Table 3-5.

A.1.1 Terminal Type

Sets the bit pump terminal type. In any operational HDSL system, a bit pump on one side of the loop should be defined to be an HTU-C (HDSL Terminal Unit—Central office side) and the bit pump on the other side should be defined to be an HTU-R (HDSL Terminal Unit—Remote). The two terminal types differ in their activation procedure, timing recovery mechanism, scrambler and descrambler taps, and more. The terminal type must be properly defined prior to any activation operation.

In a multipair system, each bit pump can be individually set as an HTU-C or HTU-R (although in most implementations, all bit pumps will be set to the same terminal type).

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x01	_TERMINAL_TYPE

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
HTU-C (default)	Central Office Terminal	0x00	_HTUC
HTU-R	Remote Terminal	0x01	_HTUR

A.1.2 Analog AGC Configuration

Provided to maintain backward compatibility with Bt8952 and Bt8958 devices. The ZipWire bit pumps always use their internal 6-level AAGC configuration.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x02	_ANALOG_AGC_CONFIG

Options			
Option	**Description**	**Data Field Value**	**C Constant (api.h)**
None	—	0x00	—

A.1.3 Startup Sequence Source

Selects the source of activation sequences to be external or internal. The bit pump generates internal activation sequences that can be used in a complete standalone transceiver implementation. The internal activation sequences are 2- or 4-level scrambled 1s, with no HDSL framing information. These sequences do not meet the standard activation requirements that require HDSL framing information to be included in the activation sequences.

If a framed activation sequence is required, it must be supplied prior to any activation operation and the startup sequence source must be set to external. Only a 4-level sequence need be supplied. The bit pump ignores the magnitude bit during the initial 2-level transmission.

NOTE: When the internal option is specified, the _TRANSMIT_EXT_DATA command must be called when the activation process is successfully completed.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x03	_STARTUP_SEQ_SOURCE

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
External Startup Sequence (default)	Startup sequences are supplied externally. A 4-level activation sequence must be supplied before activation is initiated.	0x00	_EXTERNAL
Internal Startup Sequence	Startup sequences are generated internally. These sequences are 2- or 4-level scrambled 1s, with no HDSL framing.	0x01	_INTERNAL

A.1.4 Transmit Scrambler

Activates or bypasses the internal transmit scrambler. The internal transmit scrambler (and receive descrambler) can be used for standalone operation of the bit pump. When activated, all incoming data is scrambled and then converted to quaternary symbols for transmission.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x04	_TRANSMIT_SCR

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Bypass (default)	The transmit scrambler is not used, and the symbols supplied by the application are transmitted with no change.	0x00	_BYPASS
Active	The transmit scrambler is activated, and the bit stream supplied by the application is scrambled on-chip before being transmitted. Standard scrambler taps are used (according to terminal type setting). The Tx scrambler is operated (when turned on) only during 4-level transmission, i.e., during 4-level activation transmission and during normal operation.	0x01	_ACTIVATE_SCR

A.1.5 Receive Descrambler

Activates or bypasses the internal receive descrambler, which can be used for standalone operation of the bit pump. When activated, all detected symbols are converted to bits and descrambled prior to being transferred to the data output pins.

NOTE: All received bits are descrambled, including framing and overhead bits.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x05	_RECEIVE_DESCR

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Bypass (default)	The receive descrambler is not used, and the detected symbols are supplied with no change	0x00	_BYPASS
Active	The receive descrambler is activated, and the received bit stream is descrambled before being transferred	0x01	_ACTIVATE_DESCR

A.1.6 Data Transfer Format

Selects the format in which data is transferred between the bit pump and the application. This option has no effect on data value; only the data transfer format is affected. Different data transfer formats allow for different schemes of framer bit pump clock distribution. For more details, see the bit pump data sheet.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x06	_FRAMER_FORMAT

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Parallel data, clock outputs	The bit pump outputs a baud rate clock signal (QCLK) that clocks the data transfer in both the receive and transmit directions. Received quats are transferred to the channel unit via the RQ[0] and RQ[1] pins. Transmitted quats are transferred to the bit pump via the TQ[0] and TQ[1] pins.	0x00	_PARALLEL_MASTER
Parallel data, clock inputs	The bit pump outputs a baud rate clock signal (RBCLK) that clocks the data transfer in the receive direction. It also outputs a separate baud rate clock (TBCLK) that clocks the data transfer in the transmit direction. Received and transmitted quats are transferred via the TQ and RQ signals, as in “Parallel data, clock outputs” mode. The RBCLK and TBCLK signals must be a derivative of the bit pump’s 16X clock at an arbitrary phase.	0x01	_PARALLEL_SLAVE
Serial swap data	The bit pump outputs a baud rate clock (on the QCLK pin) and a bit rate clock (on the RQ[0] pin) that clocks the data transfer in both receive and transmit directions. The received and transmitted quats are transferred serially, each on a single line, via the RQ[1] and TQ[1] pins. The 2B1Q magnitude bit is aligned to QCLK low, and the 2B1Q sign bit is aligned to QCLK high.	0x02	_SERIAL

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Serial swap data	The bit pump outputs a baud rate clock (on the QCLK pin) and a bit rate clock (on the RQ[0] pin) that clocks the data transfer in both receive and transmit directions. The received and transmitted quats are transferred serially, each on a single line, via the RQ[1] and TQ[1] pins. The 2B1Q magnitude bit is aligned to QCLK high, and the 2B1Q sign bit is aligned to QCLK low. This format satisfies the ETSI/ANSI requirements for output quat orientation.	0x03	_SERIAL_SWAP

A.1.7 Other Side Conexant

In a system where both terminals use Conexant bit pumps, several activation operations can be performed more efficiently relative to the standard requirements. This command informs the ZipWire software that the other terminal uses a Conexant bit pump.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x07	_BT_OTHER_SIDE

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Other side unknown (default)	The terminal on the other loop end is not known to use a Conexant bit pump	0x00	_NO_BT
Other side Conexant	The terminal on the other side of the loop uses a Conexant bit pump	0x01	_BT

A.1.8 LOST Time Period

The LOS timer is restarted when a LOS condition is detected and the bit pump ASM is in the DEACTIVE_MODE. When this timer reaches a predefined value, the LOST status bit is turned on. Once turned on, the status bit is not cleared (even if there is no longer a LOS condition). The LOST indication is cleared only when an Activate or Reset command is issued.

The Deactivate API command places the bit pump ASM in the DEACTIVE_MODE, which is one condition that enables the LOST mechanism. Thus, during activation or normal operation, the LOST status is never set. This implementation is in agreement with ANSI and ETSI specifications. The LOST time interval is programmable in the range of 0–31 seconds with a resolution of 1/10 second. The value of the LOST status bit can be checked using the Bit Pump Status command.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x08	_LOST_TIME_PERIOD

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
LOST time period	1-byte unsigned integer x. The LOST time period is set to x/10 seconds. Default Value: x = 1 second.	x	—

A.1.9 Bit Pump On/Off

Turns a bit pump on and off. Setting a bit pump state to off causes the ZipWire software to put the chip in a power-down mode and ignore any further control commands issued to this bit pump, other than bit pump on/off.

All bit pumps intended to be activated should be turned on prior to any other control operation. The bit pumps are numbered from 0–5. Any bit pump can be used in a system.

NOTE: A hardware implementation may include, for example, three bit pumps, only two of which are turned on.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x09	_SYSTEM_CONFIG

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Off (default)	Turn bit pump off	0x00	_NOT_PRESENT
On	Turn bit pump on	0x01	_PRESENT

A.1.10 Transmit External Data

Is required when the activation sequence source is set to internal by the _STARTUP_SEQ_SOURCE command. When issued, this command causes the bit pump to start transmission of externally supplied data symbols. This command should be issued upon the successful completion of activation, which is determined by the application ASM. The transmitted data should be supplied to the bit pump prior to issuing this command. Avoid transmitting a long stream of constant value symbols.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x0A	_TRANSMIT_EXT_DATA

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
None	—	0x00	—

A.1.11 Activate

Initiates the activation process. On an HTU-C terminal, this command begins a 2-level activation sequence transmission. On an HTU-R terminal, this command causes the bit pump to wait for the detection of an incoming signal and continue with the activation process when that signal is detected. The activation process itself is fully automatic. An application may inquire about the activation status, NMR, etc., but no other actions are required.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x0B	_ACTIVATE

Options			
Activation Mode	**Description**	**Data Field**	**C Constant (api.h)**
Cold Startup	Attempt a cold startup. Either ZipStartup software is not available or ZipStartup coefficients are not saved from a previous startup.	0x00	_ACTIVATE_COLD
ZipStartup	Attempt a ZipStartup. The ZipStartup software is available and the ZipStartup coefficients are saved from a previous startup.	0x01	_ACTIVATE_ZIP

A.1.12 Deactivate

Turns the transmitter off and stops all activation operations regardless of the current bit pump status. The bit pump ASM is set to the IDLE state, where it awaits further commands. The combination of the Deactivate command and a LOS condition enables the LOST (see Section A.1.8). If issued during normal operation, the receiver continues to function properly (useful for test purposes).

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x0C	_DEACTIVATE

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
None	—	0x00	—

A.1.13 Test Mode

Operates the bit pump in special test modes. These modes include maintenance and loopback modes. To execute a bit pump self-test procedure, use the Self-Test Status Request command. To turn off any of the special test modes, use the Test Mode command with an _EXIT_TEST_MODE (0 value) parameter. All test modes except the two digital loopback modes require, after exiting the test mode, that a complete activation procedure be repeated (assuming normal operation is required). When exiting all these test modes (by issuing the Test Mode command with a parameter value of 0), the bit pump is initialized to a reset state and goes to the IDLE state, where it awaits further commands.

The two digital loopback test modes do not interfere with the operation of the bit pump and only affect the received/transmitted symbol stream. Therefore, operating the bit pump in a digital (near/far) loopback during normal operation does not cause loss of synchronization. Exiting one of these two test modes turns off the loopback operation, allowing for normal data transfer to continue with no need for a restart. Table A-1 summarizes the action taken when exiting different test modes.

NOTE: When exiting a Digital Far Loopback mode, the transmitted symbol sequence source depends on the setting of the startup sequence source (see Section A.1.3).

The analog loopback test modes activate the test mode state machine in *_HandleTestMode()*. The application ASM reads the normal_operation and activation_interval bits returned by the _STARTUP_STATUS API command to determine when the test mode has successfully completed or failed (respectively).

After the application code invokes the specified Test Mode API command, the application code must repeatedly call *_BtMain()* and query the _STARTUP_STATUS API command to determine when the test is completed. This is the same concept as when activating the bit pump startup process.

The isolated pulse tests measure pulse templates. The 2- and 4-level tests measure transmit signal power.

Table A-1. Exiting Test Modes

Test Mode	Action Taken When Exiting Test Mode	How to Go Back to Normal Data Transfer(1)
Digital Near Loopback	Rx symbol stream (bit pump-to-framer) is sent back to be the detected symbols sequence.	Exit test mode.
Digital Far Loopback	External activation sequence mode: Transmit external data (Tx scrambler is operated and bypassed according to the Transmit Scrambler Mode setting).	Exit test mode.
—	Internal activation sequence mode: Transmit internally generated 4-level scrambled 1s sequence.	Exit test mode. Issue Transmit External Data command to start transmitting the externally supplied (payload) data.
All other test modes	Bit pump initialized to a reset state, and goes to an IDLE state.	Complete activation operation is required.

NOTE(S):
(1) Assuming normal data transfer took place before issuing test mode.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x0D	_TEST_MODE

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Exit Test Mode	Cancel test mode. See Table A-1.	0x00	_EXIT_TEST_MODE
External Analog Loopback	Transmit the externally supplied Tx symbols. Use the echo signal as a received signal and detect the symbols using the standard equalizer.	0x01	_ANALOG_LOOPBACK
Digital Near Loopback	Tx symbols supplied to the bit pump by the framer are looped back as the Rx symbols going from the bit pump to the framer. Useful for testing the framer and the framer and bit pump connection.	0x02	_NEAR_LOOPBACK
Digital Far Loopback	Detected Rx symbols are transmitted back on the loop. Useful for testing full 2-way transmission over a loop.(1)	0x03	_FAR_LOOPBACK
Transmit Isolated +3 Pulse	Transmit (repeatedly) an isolated +3 level pulse. Useful for testing the transmitted pulse shape.	0x04	_ISOLATED_PULSE_PLUS3
Transmit Isolated +1 Pulse	Transmit (repeatedly) an isolated +1 level pulse.	0x05	_ISOLATED_PULSE_PLUS1
Transmit Isolated –1 Pulse	Transmit (repeatedly) an isolated –1 level pulse.	0x06	_ISOLATED_PULSE_MINUS1
Transmit Isolated –3 Pulse	Transmit (repeatedly) an isolated –3 level pulse.	0x07	_ISOLATED_PULSE_MINUS3
Continuous 4-level Transmission	Continuous transmission of a 4-level scrambled 1s sequence (internally generated). Useful for measuring transmitted power and spectral shape.	0x08	_FOUR_LEVEL_SCR
Continuous 2-level Transmission	Continuous transmission of a 2-level scrambled 1s sequence (internally generated).	0x09	_TWO_LEVEL_SCR
Set Nominal VCXO Frequency	Set VCXO control word to its nominal value. Useful for measuring VCXO center frequency.	0x0A	_VCXO_NOMINAL

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Set Minimum VCXO Frequency	Set VCXO control word to its minimum value.	0x0B	_VCXO_MIN
Set Maximum VCXO Frequency	Set VCXO control word to its maximum value.	0x0C	_VCXO_MAX
Internal Analog Loopback (Transmitting Loopback)	Transmits the externally supplied Tx symbols out the TxP and TxN pins and detects the symbols on the hybrid inputs (RxBP and RxBN); the receive inputs (RxP and RxN) are bypassed.	0x0D	_INTERNAL_ANALOG_LOOPBACK
Isolated Analog Loopback (Silent Loopback)	The externally supplied Tx symbols are internally looped back in the bit pump. The transmitter (TxP and TxN) is turned off (silent).	0x0E	_ISOLATED_ANALOG_LOOPBACK
ERLE Test	Test the analog and digital echo canceler performance.	0x0F	_ERLE_TEST
AAGC Test	Test the AAGC. The test is run with the transmitter in either 2- or 4-level scrambled 1s mode based on the _ERLE_TEST_MODE transmit level configuration.	0x10	_MEASURE_AAGC
Alternating Symbol +3/–3	Transmits (repeatedly) a square wave of alternating +3/–3 symbols. The duty cycle is dependent on the meter interval register (0x18, 0x19) which defaults to 0x0400.	0x11	_ALTERNATING_SYMBOLS_3
Alternating Symbol +1/–1	Transmits (repeatedly) a square wave of alternating +/–1 symbols. The duty cycle is dependent on the meter interval register (0x18, 0x19) which defaults to 0x0400.	0x12	_ALTERNATING_SYMBOLS_1

NOTE(S):
(1) The operation of a digital far loopback requires activating the bit pump internal Tx scrambler and Rx descrambler.

A.1.14 Symbol Rate (_SYM_RATE and _SYM_RATE_HI)

Informs the ZipWire software of the bit pump symbol rate. This value must be programmed for proper bit pump operation.

The Symbol Rate parameter is determined by either of these equations:

$$x = \text{Symbol Rate} \,/\, 4000$$

or

$$x = \text{Data Rate} \,/\, 8000$$

The supported range of x is 18 (data rate = 144 Kbps) to 290 (data rate = 2,320 Kbps).

To program the desired symbol rate, the _SYM_RATE_HI command must be issued before the _SYM_RATE command. The device is programmed when the _SYM_RATE command is received. The _SYM_RATE_HI parameter should be programmed to 0 with the Bt8960 and Bt8970 devices.

For the Bt8960 and Bt8970, the symbol rate was primarily used to convert seconds to symbols for timers within the bit pump. In the RS8973, the symbol rate sets both the timer intervals and the frequency synthesizer registers.

NOTE: If the _SYM_RATE command is issued, it must be called immediately after the _SYSTEM_CONFIG ON command. The _SYM_RATE command sets the PLL Clock Center Frequency. There is a potential for the bit pump registers to become corrupted when the PLL Clock Frequency is changed. The _SYM_RATE command initializes the bit pump registers to their default values after writing the PLL Clock Center Frequency. Therefore, the user must issue their application-specific API command values after the _SYM_RATE command is issued.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x0E	_SYM_RATE

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Symbol Rate	Contains the lower 8 bits of the Symbol Rate parameter x where x = Symbol Rate / 4,000 Default Value: 98 = 392 Kbps / 4,000	x	—

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x22	_SYM_RATE_HI

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Symbol Rate High	Contains the upper 2 bits of the Symbol Rate parameter x	x	—

A.1.15 Reset

Loads the bit pump microcode, resets all bit pump internal registers, and sets all user-programmable options to their default values. After issuing a reset, all non-default user-programmable options should be reprogrammed using the appropriate API commands. The Reset operation is automatically performed when turning a bit pump on using the bit pump on/off command. Therefore, under normal operating conditions, the application code does not need to call the Reset command.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x0F	_RESET_SYSTEM

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
None	—	0x00	—

A.1.16 Operate NonLinear Echo Canceler (NLEC)

Selects between operating or disabling the bit pump NLEC. Conexant recommends not using the NLEC.

NOTE: The NLEC is not functional in Bt8960, Revisions A and B, and should not be activated.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x11	_OPERATE_NLEC

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Disable NLEC	NLEC is not activated and not used	0x00	_NLEC_OFF
Enable NLEC	NLEC operates normally	0x01	_NLEC_ON

A.1.17 Write Transmitter Gain

Writes the Transmitter Gain Register (0x29). The Transmitter Gain register is a 4-bit, 2s complement value. The upper 4 bits of this field are ignored. The Tx Gain adjusts the nominal transmit power of the bit pump. The Tx Gain ranges from –1.6 dBm (1000b) to +1.4 dBm (0111b) of the nominal transmit power level.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x13	_WRITE_TX_GAIN

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Tx Gain	4-bit, 2s complement integer	Value	—

A.1.18 Tip/Ring Reversal

Reverses the tip/ring polarity on the received signal. The bit pump does not reverse the tip/ring polarity on the transmitted signal.

This command is useful in applications where a framer has the ability to detect tip/ring reversal but cannot correct it. Because this command only reverses the received signal, it is necessary to call this command on both the Central and Remote terminals when tip/ring reversal is detected.

NOTE: The bit pump does provide the ability to detect and correct tip/ring reversal under special conditions (see Section A.1.21).

Opcode	
Numeric Value	**Opcode**
0x14	_REVERSE_TIP_RING

Options			
Option	**Description**	**Data Field**	**C Constant (defined in file api.h)**
Tip/Ring Normal	Sets the tip/ring polarity on the received signal to normal (not reversed)	0x00	_TIP_RING_NORMAL
Tip/Ring Reverse	Reverses the tip/ring polarity on the received signal	0x01	_TIP_RING_REVERSE

A.1.19 BER Meter Start

Activates the BER meter. The bit pump is set to transmit an internal 4-level scrambled 1s pattern. The enabled bit is set, and the *bit_errors* and *meter_intervals* variables are reset to 0. This command should only be called when the bit pump is in normal operation.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x15	_BER_METER_START

Options			
Option	**Description**	**Data Field**	**C Constant (defined in file api.h)**
None	—	0x00	—

A.1.20 BER Meter Stop

Deactivates the BER meter. The bit pump is set to transmit external, 4-level data. The transmit scrambler is set based on the current _TRANSMIT_SCR API setting. The enabled bit is turned off. Because the *bit_errors* and *meter_intervals* variables are unmodified, they remain valid.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x16	_BER_METER_STOP

Options			
Option	**Description**	**Data Field**	**C Constant (defined in file api.h)**
None	—	0x00	—

A.1.21 Auto Tip/Ring Reversal

Enables or disables the automatic tip/ring detection and reversal of the DSL. When enabled, the bit pump activation procedure automatically detects and corrects tip/ring reversal. For tip/ring reversal to work, the user must set the internal scrambled 1s during activation, enable the scrambler and descrambler, and have a Conexant bit pump on the far end configured in the same fashion (none of these requirements are handled by this API command). Once enabled, the *_HturControlProcess()* and *_HtucControlProcess()* functions check for tip/ring reversal and, if it occurs, they reverse the data receiver.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x17	_AUTO_TIP_RING

Options			
Option	**Description**	**Data Field**	**C Constant (defined in file api.h)**
Disable	Disables the Auto Tip/Ring Reversal	0x00	_AUTO_TIP_RING_OFF
Enable	Enables the Auto Tip/Ring Reversal	0x01	_AUTO_TIP_RING_ON

A.1.22 Background and ERLE Test Mode

Activates the ERLE test mode. This command forces the bit pump ASM into the IDLE state. To abort the ERLE test mode before completion, use the _TEST_MODE command (with _EXIT_TEST_MODE option).

This command is similar to the _TEST_MODE command (with _ERLE_TEST option) but provides more flexibility because it includes a parameter byte that configures the test. The _TEST_MODE command (with _ERLE_TEST option) invokes the test with the current ERLE setup options.

NOTE: The Background Noise Test and ERLE test are typically run with the NLEC bypassed, transmitter set to 4-level, and AAGC set to 12 dB.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x18	_ERLE_TEST_MODE

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
—	See Table A-2	1-byte	—

Table A-2. ERLE Configuration Byte

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NLEC Mode	Transmit Level	AGAIN2	AGAIN1	AGAIN0	Reserved	Reserved	Transmit State

NLEC Mode	Description
0	Do not adapt the NLEC
1	Adapt the NLEC

Transmit Level	Description
0	Set transmitter to 4-level
1	Set transmitter to 2-level

AGAIN2	AGAIN1	AGAIN0	Gain Setting
0	0	0	0 dB
0	0	1	3 dB
0	1	0	6 dB
0	1	1	9 dB
1	0	0	12 dB
1	0	1	15 dB

The 3 AGAIN bits select the AGAIN settings. These correspond to the 3 Gain Control bits of the ADC Control register (0x21).

Transmit State	Description
0	Disables the transmitter, used to measure Background Noise Test
1	Enables the transmitter, used to measure ERLE

A.1.23 Update ZipStartup Coefficients

Updates the bit pump's echo canceller and equalizer coefficients. This command should only be called during normal operation.

Opcode	
Numeric Value	C Constant (defined in file api.h)
0x19	_ZIP_START_UPDATE

Options			
Option	Description	Data Field	C Constant (defined in file api.h)
None	—	0x00	—

A.1.24 ZipStartup Configuration

Enables or disables the ZipStartup code. This command allows the ZipStartup code to be bypassed even though the ZIP_STARTUP compiler directive is enabled.

When enabled, the bit pump algorithm supports the ZipStartup code (i.e., save coefficients during the activation, allow coefficients to be updated during normal operation, etc.). Disabling the ZipStartup forces all activation attempts in the cold mode.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x20	_ZIP_START_CONFIG

Options			
Option	**Description**	**Data Field**	**C Constant (defined in file api.h)**
Off	Disable ZipStartup	0x00	—
On	Enable ZipStartup	0x01	—

A.1.25 HCLK Select

Selects the HCLK pin output frequency.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x21	_HCLK_SELECT

Options			
Option	**Description**	**Data Field**	**C Constant (defined in file api.h)**
Default	Default Frequency: 16 times QCLK (Bt8970 and RS8973) 64 times QCLK (Bt8960)	0x00	_HCLK_DEFAULT
16x	16 times QCLK	0x01	_HCLK_16_TIMES
32x	32 times QCLK	0x02	_HCLK_32_TIMES
64x	64 times QCLK	0x03	_HCLK_64_TIMES

A.1.26 Activation Time-Out

Sets the activation interval time-out setting. An invalid data field parameter sets the activation time-out to _ACT_TIME_30SEC. This timer runs within the bit pump and is polled by the *_HandleFlags()*. When the timer expires, *_HandleFlags()* sets the activation_timeout flag which the application ASM checks during activation.

The variable number of symbols option uses a linear formula that calculates the number of symbols based on data rate. Table A-3 lists time-out values for common data rates. The formula was derived by setting the time-out to equal approximately twice the typical startup times.

Table A-3. Activation Time-Out Values (_ACT_TIME_VARIABLE Option)

Data Rate (Kbps)	Typical Startup Time (s)	Activation Time-Out (s)
144	64.4	128.8
288	35.3	70.5
416	26.3	52.6
784	16.8	33.7
1,168	13.3	26.7
1,552	11.5	23.1
2,320	9.8	19.5

The *_SetActivationInterval()* function in user.c sets the activation time-out based on the _ACTIVATION_TIMEOUT API command setting. For certain applications, these time-out values may need to be modified depending on the microprocessor loading. For example, in a 24-bit pump system, the startup times may take longer if all the bit pumps are activated simultaneously. In this scenario, the time-out values must be increased.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x23	_ACTIVATION_TIMEOUT

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Fixed 30 second (default)	The Activation Interval is set to 30 seconds. This option should be used when running at the standard HDSL data rates.	0x00	_ACT_TIME_30SEC
Fixed 52 second	The Activation Interval is set to 52 seconds. This option is used when running at 288 Kbps data rate.	0x01	_ACT_TIME_52SEC
Variable number of symbols	The Activation Interval is based on data rate rather than fixed time. This option should be used in rate-adaptive systems. See Table A-3.	0x02	_ACT_TIME_VARIABLE

A.1.27 Regenerator Mode

Enables the RS8973 for Regenerator mode. This mode bypasses the internal PLL and timing recovery circuitry. The symbol rate then equals MCLK / 16. This command should only be used by the HTU-C in a regenerator system. Only the RS8973 device supports this command.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x24	_REGENERATOR_MODE

Options			
Option	**Description**	**Data Field**	**C Constant (defined in file api.h)**
Disable	Disable the regenerator mode. Uses internal PLL clock synthesizer and timing recovery.	0x00	_REGENERATOR_OFF
Enable	Enable the regenerator mode.	0x01	_REGENERATOR_ON

A.2 Status Request Commands

Provide performance monitoring information from the bit pump. They do not affect the bit pump operation in any way. The bit pump response to all status-request commands is a 1-byte response.

For each status-request command, the type and format of information supplied, the command opcode, and the options are described.

A.2.1 Input Signal Level

Requests the level of the average signal level at the ADC input.

NOTE: The signal at the ADC input consists of a large, transmitted echo component, and a smaller, far-end signal component. Thus, cable attenuation data cannot be extracted from this information.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x80	_SLM

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
None	—	0x00	—

Bit Pump Response

Unsigned integer x, $0 \le x \le 255$, relative to the average absolute value of the ADC input signal. The measurement scale is such that a value of 255 corresponds to the ADC positive full-scale value.

A.2.2 Input DC Offset

Requests the value of the average DC level at the ADC input.

NOTE: Any level of input DC offset is digitally canceled on-chip, but large DC offsets (>2% of full scale) may degrade performance because of the reduction in effective ADC dynamic range.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x81	_DC_METER

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
None	—	0x00	—

Bit Pump Response

Signed integer x, $-128 \le x \le 127$, relative to the average DC offset per ADC sample. The measurement scale is such that the actual DC offset in units of ADC LSB is $32x$. In case the actual DC offset is outside the representable range (–4096 to 4095), the nearest representable value is used.

A.2.3 Far-End Signal Attenuation

Requests a value of the Far-End Level Meter (FELM). This value is based on measuring the average far-end signal level after echo cancellation. The FELM measurement is automatically scaled for the different AAGC settings.

The result is calibrated to represent the overall signal power attenuation over the cable in dB, and is calibrated for the nominal 13.5 dBm transmit power at the far-end.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x82	_FELM

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
None	—	0x00	—

Bit Pump Response

A 1-byte unsigned integer x, $0 \le x \le 255$, indicating the overall signal power attenuation, in units of 1.0 dB.

A.2.4 Noise Margin

Requests a Noise Margin Reading (NMR). The noise margin is defined as the maximum tolerable increase in external noise power that still allows for a BER of less than 1^{-7}.

The value is based on measuring the average absolute level of the noise at the input to the slicer. To reduce the statistical measurement error of each sample, it is recommended that the average of 10 noise margin readings be taken.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x83	_NMR

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
None	—	0x00	—

Bit Pump Response

A 1-byte signed integer x, $-16 \le x \le 127$, indicating the noise margin in units of 0.5 dB. For example, a value of –8 means a noise margin of –4 dB.

A.2.5 Timing Recovery Control

Requests a value of the Timing Recovery Control circuit. This value indicates the timing recovery frequency relative to its center frequency, which does not necessarily equal the nominal transmission frequency. On an HTU-C terminal, this response is always zero because the control circuit is set to its nominal value. On an HTU-R terminal, this value gives an estimate of the frequency offset relative to the center frequency. The relation of the given value with the frequency offset in Hertz depends on the timing recovery’s slope.

NOTE: This replaces the _VCXO_CONTROL_VOLTAGE API command found in previous HDSL products.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x84	_TIMING_RECOVERY_CONTROL

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
None	—	0x00	—

Bit Pump Response

A signed integer x, $-128 \le x \le 127$, indicating the 8 MSBs of the timing recovery control word.

A.2.6 Bit Pump Status

Requests bit pump status. The status bits are designated I0 (LSB) to I7 (MSB), and interpreted according to Table A-4.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x85	_STARTUP_STATUS

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
None	—	0x00	—

Bit Pump Response

A status byte (see Table A-4).

Table A-4. Status Bits

Status Bit	Indicates	Value = 0	Value = 1
0 (LSB)	LOS: A LOS condition is defined as the absence of a far-end signal at the receiver input.	No LOS condition	LOS condition
1	LOST: A LOST condition is defined as a continuously active LOS condition for more than a period of LOST seconds. The LOST period is user-programmable (see the LOST Time Period command).	No LOST condition	LOST condition
2	Tip/Ring Indication: This bit is set whenever the bit pump receive polarity is set.	Tip/Ring not reversed	Tip/Ring reversed
3	Activation timer: On an HTU-R, the activation timer is activated when 2-level transmission begins. On an HTU-C, the activation timer is activated when an Activate command is issued and 2-level transmission begins. The timer is restarted when a far-end (HTU-R) signal is detected(1).	Timer not expired	Timer expired
4	Noise Margin OK: Indicates that the noise margin is better than a threshold of –5 dB.	NMR ≤ –5dB	NMR > –5 dB
5	Run LOST timer: Indicates whether or not the LOS timer is running. The timer uses the sut3 timer in the bit pump.	Timer running	Timer not running
6	Transmitter 4-level Indicator: Indicates that a 4-level signal is being transmitted.	Not transmitting 4-level	Transmitting 4-level
7	Normal Operation Flag: This bit is set when the bit pump successfully completes startup. This bit is cleared when the startup is first activated, when the Deactivate command is issued, and when the software is initialized.	Not normal operation	Normal operation

NOTE(S):
(1) The activation timer defaults to 30 seconds.

A.2.7 Read Linear Echo Canceler (LEC) Coefficient

Reads a specified LEC coefficient.

NOTE: Provided for internal debugging; not required in customer applications.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x86	_LEC_COEFF

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Coefficient index	For the Bt8960 device: Unsigned integer ×, 0 ≤ × ≤ 59, indicating the index of the LEC coefficient to be read. For the Bt8970 and RS8973 devices: Unsigned integer ×, 0 ≤ × ≤ 119, indicating the index of the LEC coefficient to be read.	×	—

Bit Pump Response

A 1-byte signed integer *x*, containing the 8 MSBs of the requested EC coefficient value.

A.2.8 Read NonLinear Echo Canceler (NLEC) Coefficient

Reads a specified NLEC coefficient.

NOTE: Provided for internal debugging; not required in customer applications.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x87	_NLEC_COEFF

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Coefficient index	Unsigned integer ×, 0 ≤ × ≤ 63, indicating the index of the NLEC coefficient to be read.	×	—

Bit Pump Response

Signed integer *x*, containing the 8 MSBs of the requested NLEC coefficient value.

A.2.9 Read EQ Coefficient

Reads a specified DAGC, FFE, or EP coefficient.

NOTE: Provided for internal debugging; not required in customer applications.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x88	_EQ_COEFF

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
FFE Coefficients	Request FFE coefficients 0–7	0–7	—
FFE Data Taps	Request FFE Data Taps 0–7	8–15	—
EP Coefficients	Request EP coefficients 0–4	16–20	—
EP Data Taps	Request EP Data Taps 0–4	21–25	—
DAGC Gain	Request DAGC gain value—LSB	26	—
	Request DAGC gain value—MSB	27	—
DAGC Output	Request DAGC Output	28	—
FFE Output	Request FFE Output	29	—
DAGC Input	Request DAGC Input	30	—
FFE Output, delayed 1 Symbol	Request FFE Output, delayed 1 Symbol	31	—
DAGC Error Signal	Request DAGC Error Signal	32	—
Equalizer Error Signal	Request Equalizer Error Signal	33	—
Slicer Error Signal	Request Slicer Error Signal	34	—
Reserved	—	35–47	—

Bit Pump Response

A signed integer containing the 8 MSBs of the requested EQ coefficient value.

A.2.10 Read DFE Coefficient

Reads a specified DFE coefficient.

NOTE: Provided for internal debugging; not required in customer applications.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x89	_DFE_COEFF

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Coefficient index	For the Bt8960 device: Unsigned integer $\times$, $0 \le \times \le 57$, indicating the index of the DFE coefficient to be read. For the Bt8970 and RS8973 devices: Unsigned integer $\times$, $0 \le \times \le 115$, indicating the index of the DFE coefficient to be read.	$\times$	—

Bit Pump Response

A signed integer *x*, containing the 8 MSBs of the requested DFE coefficient value.

A.2.11 Software and Chip Version

Returns the software and chip version numbers.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x8A	_VERSION

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Major SW Version Chip Version	The 4 LSBs of the status byte (bits 0–3) contain the bit pump major software version, bit 0 being the LSB. The 4 MSBs of the status byte (bits 4–7) indicate the bit pump chip version, bit 4 being the LSB.	0x00	_HW_SW_VERSIONS
Major SW Version	1-byte unsigned integer field returning the major software version.	0x01	_MAJOR_SW_VERSION
Minor SW Version	1-byte unsigned integer field returning the minor software version.	0x02	_MINOR_SW_VERSION
Bit Pump Type Chip Version	The 3 LSBs of the status byte (bits 0–2) contain the bit pump type (see table below). Bit 3 is undefined. The 4 MSBs of the status byte (bits 4–7) indicate the bit pump chip version, bit 4 being the LSB.	0x03	_HW_TYPE_VERSIONS
ZipStartup Major SW Version	1-byte unsigned integer field returning the major software version.	0x04	_ZIP_MAJOR_VERSION
ZipStartup Minor SW Version	1-byte unsigned integer field returning the minor software version.	0x05	_ZIP_MAJOR_VERSION

Bit Pump Types	
Bits 2–0	**Bit Pump Type**
000	Bt8952
001	Bt8960
010	Bt8970
011	RS8973
100–111	Undefined

A.2.12 Bit Pump Present

Returns a bit pump presence status. A positive response means the bit pump is present in the system and connected to the microprocessor. The presence of a bit pump is determined by performing a write/read operation to a single internal bit pump RAM location. It is possible for hardware faults to masquerade as the presence of a bit pump at a location. The presence of a bit pump does not guarantee that all microprocessor bit pump hardware connections are valid.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x8B	_BIT_PUMP_PRESENT

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
None	—	0x00	—

Bit Pump Response

- 0x00 = Bit pump not present
- 0x01 = Bit pump present

A.2.13 Self-Test

Executes a bit pump self-test. The self-test does not cover all aspects of the chip operation, but verifies read/write operations.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x8C	_SELF_TEST

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
None	—	0x00	—

Bit Pump Response

- 0x00 = Bit pump self-test pass
- 0x01 = Bit pump self-test fail

A.2.14 Read Bit Pump Register

Reads an internal bit pump register. The command must specify a register address.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x8D	_REGISTER

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Bit pump register address	Integer *x* is the address of the internal bit pump register to be read. See *RS8973 Single Chip SDSL/HDSL Transceiver Data Sheet* (100nnnx, formerly N8973DSD) for register address map.	×	—

Bit Pump Response

Value stored in specified bit pump internal register.

A.2.15 Bit Pump Configuration

Returns a bit pump configuration, including user-programmable options and parameters.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x8E	_CONFIGURATION

Option			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Read setup low byte	Read low byte of user setup. See Table A-5 for the structure of the user setup, low byte.	0x00	_USER_SETUP_LOW_BYTE
Read setup high byte	Read high byte of user setup. See Table A-6 for the structure of the user setup, high byte.	0x01	_USER_SETUP_HIGH_BYTE
Read LOST time period	Read LOST time period. The format of this status byte is identical to the format in the LOST Time Period command.	0x02	_LOST
Read symbol rate low value	Read symbol rate. Returns the lower 8 bits of the 10-bit symbol rate setting.	0x03	_BIT_RATE
Read ERLE setup byte	Read the ERLE configuration byte. See Table A-2 (see Section A.1.22) for the structure of application setup word.	0x04	_ERLE_SETUP_CONFIG
Read symbol rate high value	Returns the upper 2 bits of the 10-bit symbol rate setting.	0x05	_BIT_RATE_HI
Read application setup, low byte	Read low byte of the application setup. See Table A-7 for the structure of the application setup, low byte.	0x06	_USER_PARAM_LOW_BYTE

Table A-5. User Setup, Low Byte

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Auto tip/ring reversal	Other side Conexant	Receive descr.	Transmit scramb.	Startup sequence source	Reserved	Reserved	Terminal type

Table A-6. User Setup, High Byte

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Test mode					Activation mode ZipStartup = 1 Cold Start = 0	Data transfer format	

All fields of the user setup word are coded in exactly the same way as the data field byte in the corresponding control command. For example, the two Data Transfer Format bits (bits 0 and 1 of the high user setup byte) are interpreted exactly the same as the data field byte in the Data Transfer Format command.

The 5 test mode bits show the current test mode index (numbered from 0–18) with test mode 0 being the LSB. A zero value indicates that no test mode is currently in effect. A non-zero value indicates that the bit pump currently operates in the specified test mode. See the Test Mode command in Section A.1.13.

Table A-7. Application Setup, Low Byte

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	Reserved	Activation Time-Out Setting		Regenerator Mode	Operate NLEC	HCLK Select	

A.2.16 State Number

Returns the software state number of different state machines within the software.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x8F	_STAGE_NUMBER

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Request Bit Pump State Number	Read the *_HturControlProcess()* or *_HtucControlProcess()* state	0x00	_ACTIVATION_STAGE
Request Temperature and Environment State Number	Read the *_HandleTempEnv()* state	0x01	_TEMP_ENV_STAGE
Request Test Mode State Number	Read the *_HandleTestMode()* state	0x02	_TEST_MODE_STAGE

Bit Pump Response

The ZipWire software internal state number. The state numbers versus state names cross mappings are defined in the files listed in Table A-8. This command is only used for testing and debugging.

Table A-8. Header Files for State Numbers and State Names

Function	Header File
_HturControlProcess() and _HtucControlProcess()	suc.h
_HandleTempEnv()	bitpump.h
_HandleTestMode()	testmode.h

A.2.17 AAGC Value

Returns the current value of the 3 AAGC control bits.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x90	_AAGC_VALUE

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
None	—	0x00	—

Bit Pump Response

The 3 LSBs of this status byte reflect the state of the AAGC control bits.

A.2.18 Read Tx Gain

Reads the Transmitter Calibration or Transmitter Gain register. The Transmitter Gain registers are 4-bit, 2s complement values. The upper 4 bits of this field are ignored.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x91	_READ_TX

Options			
Option	**Description**	**Data Field**	**C Constant (api.h)**
Tx Calibration	Transmitter Calibration register	0x00	_CALIBRATION
Tx Gain	Transmitter Gain register	0x01	_GAIN

Bit Pump Response

The Tx Calibration register contains the nominal setting for the transmitter gain. The Tx Gain register contains the current transmitter gain setting. On software initialization, the Tx Gain register is programmed to equal the Tx Calibration register.

A.2.19 BER Meter Status

Returns the BER meter status. Reading the BER meter status commands while the BER meter is enabled does not effect the BER meter operation. See Section 8.4 for details.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x92	_BER_METER_STATUS

Options			
Option	**Description**	**Data Field**	**C Constant (defined in file api.h)**
BER Status	Read the BER meter status; see the BER status below for bit definitions	0x00	_BER_STATUS
# Bit Errors Low Byte	Reads the low byte of the number of bit errors	0x01	_BER_BIT_ERRORS_LOW
# Bit Errors High Byte	Reads the high byte of the number of bit errors	0x02	_BER_BIT_ERRORS_HIGH
# Meter Intervals Low Byte	Reads the low byte of the number of meter intervals elapsed	0x03	_BER_METER_INTERVALS_LOW
# Meter Intervals High Byte	Reads the high byte of the number of meter intervals elapsed	0x04	_BER_METER_INTERVALS_HIGH

BER Status Bits			
Status Bit	**Indicates**	**Value = 0**	**Value = 1**
0	BER meter enabled	Not Active	Active
1–7	Reserved	—	—

A.2.20 ERLE Results

Queries for the Background and ERLE Test Mode results. Both the Background and ERLE Test execute the same code (the only difference is transmitter off vs. transmitter on) and thus return the same result fields. However, when analyzing the fields, only certain values are valid, as listed in Table A-9. See Section 8.5 for details.

Table A-9. Meaningful Values Returned for Different Tests

Test	Meaningful Values
Background	Only SLM and FELM
ERLE Test	All results

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x93	_ERLE_RESULTS

Options			
Option	**Description**	**Data Field**	**C Constant (defined in file api.h)**
SLM Low	SLM low byte	0x00	_ERLE_SLM_LOW
SLM High	SLM high byte	0x01	_ERLE_SLM_HIGH
FELM Low	FELM low byte	0x02	_ERLE_FELM_LOW
FELM High	FELM high byte	0x03	_ERLE_FELM_HIGH
SLM2 Low	SLM2 low byte, bypassing hybrid input	0x04	_ERLE_SLM2_LOW
SLM2 High	SLM2 high byte, bypassing hybrid input	0x05	_ERLE_SLM2_HIGH
DC Offset Low	DC offset low byte	0x06	_ERLE_DC_OFFSET_LOW
DC Offset High	DC offset high byte	0x07	_ERLE_DC_OFFSET_HIGH

NOTE: Use the following formula to convert the low and high bytes into a 16-bit value:

16-bit value = (high byte << 8) + (low byte)

The ERLE and Analog ERLE measurements are determined by the following formulas:

$$ERLE = 20 \times \log\left(\frac{SLM}{FELM}\right)$$

$$AERLE = 20 \times \log\left(\frac{SLM2}{SLM}\right)$$

A.2.21 Measure AAGC Results

Queries for the Measure AAGC Test Mode results.

Opcode	
Numeric Value	**C Constant (defined in file api.h)**
0x94	_AAGC_RESULTS

Options			
Option	**Description**	**Data Field**	**C Constant (defined in file api.h)**
SLM 0 Low	SLM @ 0 dB setting low byte	0x00	_AAGC_SLM_0_LOW
SLM 0 High	SLM @ 0 dB setting high byte	0x01	_AAGC_SLM_0_HIGH
SLM 3 Low	SLM @ 3 dB setting low byte	0x02	_AAGC_SLM_3_LOW
SLM 3 High	SLM @ 3 dB setting high byte	0x03	_AAGC_SLM_3_HIGH
SLM 6 Low	SLM @ 6 dB setting low byte	0x04	_AAGC_SLM_6_LOW
SLM 6 High	SLM @ 6 dB setting high byte	0x05	_AAGC_SLM_6_HIGH
SLM 9 Low	SLM @ 9 dB setting low byte	0x06	_AAGC_SLM_9_LOW
SLM 9 High	SLM @ 9 db setting high byte	0x07	_AAGC_SLM_9_HIGH
SLM 12 Low	SLM @ 12 dB setting low byte	0x08	_AAGC_SLM_12_LOW
SLM 12 High	SLM @ 12 dB setting high byte	0x09	_AAGC_SLM_12_HIGH
SLM 15 Low	SLM @ 15 dB setting low byte	0x0A	_AAGC_SLM_15_LOW
SLM 15 High	SLM @ 15 dB setting high byte	0x0B	_AAGC_SLM_15_HIGH

NOTE: Use the following formula to convert the Low and High bytes into a 16-bit value:

16-bit value = (high byte << 8) + (low byte)

The AAGC measurements are normalized to the 0 dB SLM reading and determined by the following formula:

$$\mathrm{AAGC}(n) = 20 \times \log\left(\frac{SLM(n)}{SLM(0)}\right)$$

where: *n* is 0 dB to 15 dB.

A.3 Acknowledge Message

The acknowledge message is sent by the 80C32 to acknowledge a correctly received message in a serial communication UI configuration. The acknowledge message is 4 bytes long (like all other messages), with the first 3 being 0xFF, 0xFF, and 0xFF. Byte 4 (the last transmitted byte) is the standard checksum byte, which equals 0x55.

Appendix B: Calibrating Noise Margin Table

B.1 Introduction

The Noise Margin Table defined in user.c has one table calibrated to an ANSI loop and one table calibrated to the W & G ILS 200 ETSI Loop 2 with 1.5 MHz-shaped noise with low crest factor. This appendix describes the steps to calibrate the noise margin table to a user's specific noise source. The Noise Margin Table has a 0.5 dB resolution.

B.1.1 Setup

Use a DSL simulator or true line and a noise generator. Select a simple standard loop like ETSI Loop 2 and perform startup. Connect a BER meter to the HDSL equipment.

B.1.2 Noise Margin Table Calibration

1. Calculate scale factor
   a. Read meter timer high byte address 0x19, using the API command:
      ```
      _BtStatus (loop_no, _REGISTER, 0x19, &scale);
      ```
   b. Calculate scale factor:

$$\text{scale\_factor} = ((2^7) / \text{scale})$$

2. Find 0 dB Noise Margin Reference Point
   a. Change the noise source until BER reading is 1.0^{-7} (for 1 loop) or 5.0^{-8} (for 2 loops). This point is the 0 dB Noise Margin reference point.
3. Calculate reference point value
   a. Read NLM meter low byte address 0x50, using the API command:
      ```
      _BtStatus (loop_no, _REGISTER, 0x50, &low_byte);
      ```
   b. Read NLM meter high byte address 0x51, using the API command:
      ```
      _BtStatus (loop_no, _REGISTER, 0x51, &high_byte);
      ```
   c. Combine the two NLM bytes (low_byte and high_byte) to an integer:

$$nlm\_value = 256 \times \text{high\_byte} + \text{low\_byte}$$

   d. Calculate the normalized value of noise level meter:

$$nlm\_value = nlm\_value \times (2^{\text{scale factor}})$$

- **e.** Repeat steps 3.a. to 3.d. ten times. Calculate the average *nlm_value* over the ten readings.

4. Set 0 dB reference point

- **a.** Write the averaged value of noise level meter to cell 32 in the *_noise_margin[]* array in user.c.

5. Calculate +0.5 dB Margin

- **a.** Decrement the noise source by 0.5 dB.
- **b.** Repeat the Calculate Reference Point Value step 3 (3.a. to 3.e.).
- **c.** Write the calculated value to the next higher cell.

6. Calculate positive margin

- **a.** Repeat step 5 (5.a. to 5.c.) until the upper half of the table is full. The significant part of the table is 0–15 dB noise margin. Higher margins cannot be measured reliably.

7. Set Noise Source to 0 dB

- **a.** Set the noise source to the 0 dB noise margin reference point.

8. Calculate –0.5 dB Margin

- **a.** Increment the noise source by 0.5 dB.
- **b.** Repeat the Calculate Reference Point Value step 3 (3.a. to 3.e.).
- **c.** Write the calculated value to the next lower cell.

9. Calculate negative margin

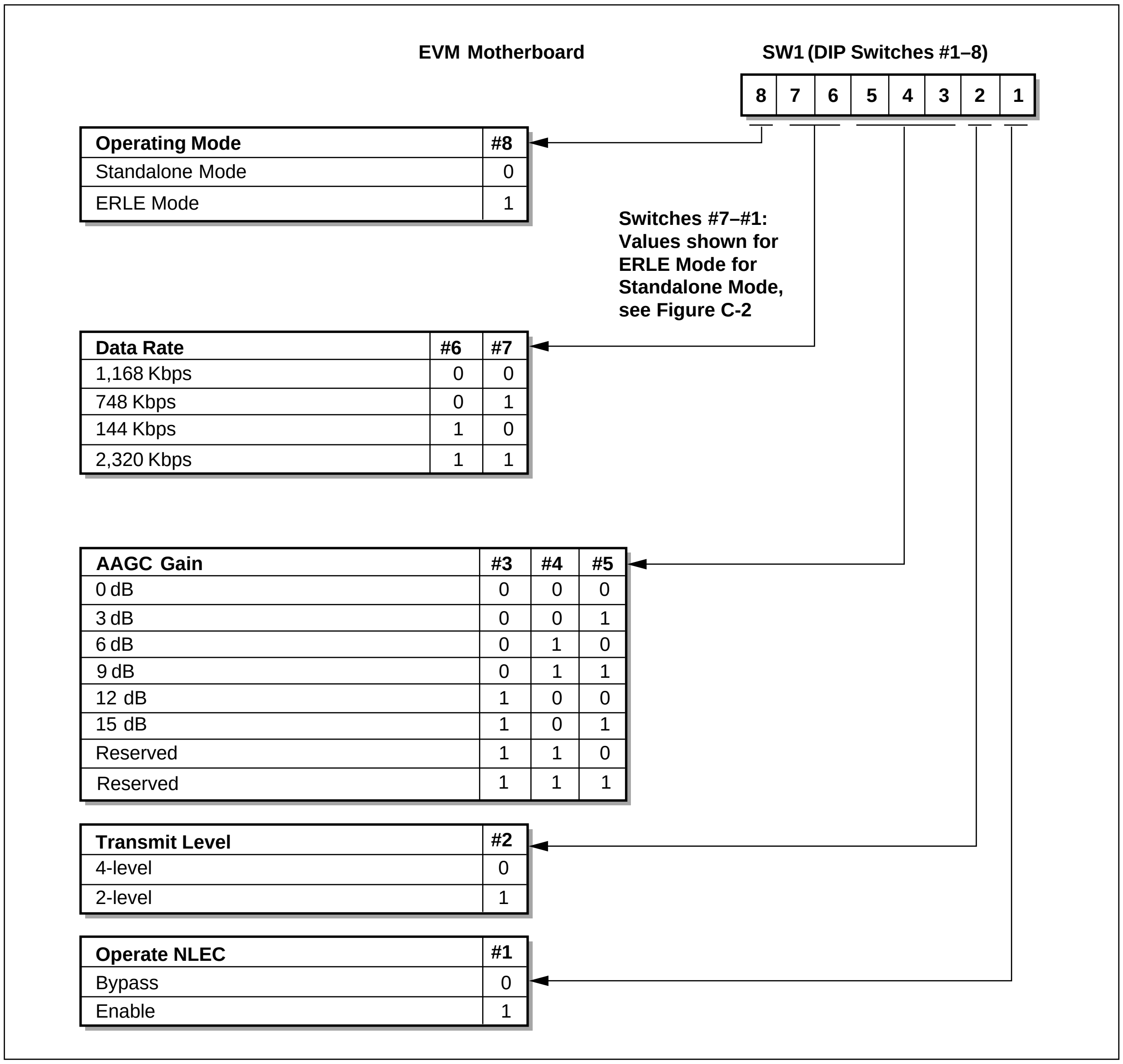
- **a.** Repeat step 8 (8.a. to 8.c.) until the lower half of the table is full.

Appendix C: EVM DIP Switch Settings

A full description of the EVM is in the *RS8973 Evaluation Module (EVM) Hardware User's Guide* (N8973UGHC). Figure C-1 and Figure C-2 illustrate the selections provided by the DIP switches on the EVM. The DIP switches select configuration parameters such as the DSL data rate and test modes. If the ZipWire software is compiled with the TDEBUG and ERLE directives defined and DIP switch #8 is set to 1, the DIP switches operate as illustrated in Figure C-1. They operate as illustrated in Figure C-2, if

- the SER_COM directive is defined, or
- the TDEBUG directive is defined and DIP switch #8 is set to 0

Figure C-1. DIP Switch Mode Configuration, TDEBUG Software, ERLE Mode


EVM Motherboard
SW1 (DIP Switches #1–8)
8 7 6 5 4 3 2 1
Switches #7–#1: Values shown for ERLE Mode for Standalone Mode, see Figure C-2
ZWIRE_019

Operating Mode	#8
Standalone Mode	0
ERLE Mode	1

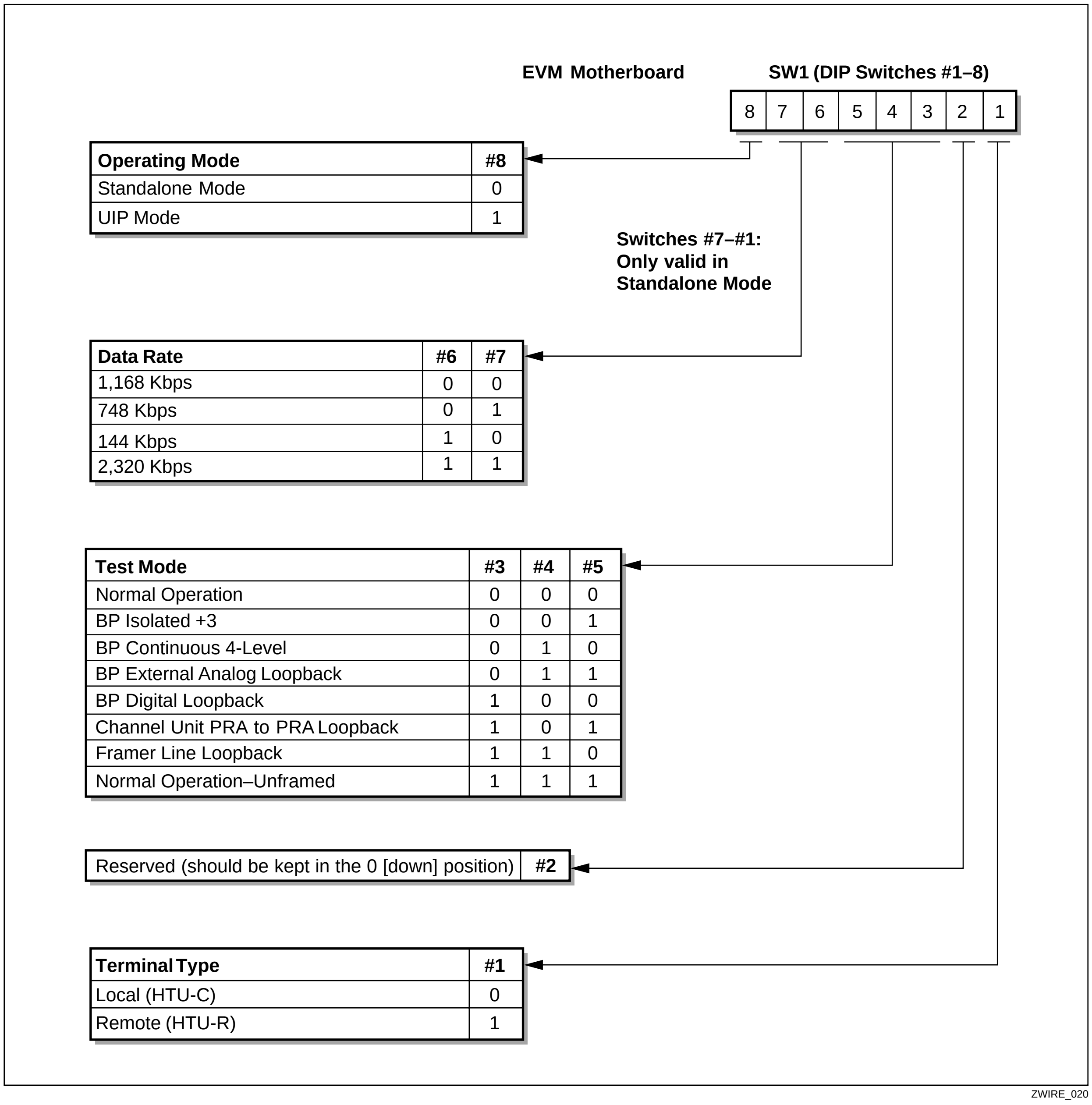
Data Rate	#6	#7
1,168 Kbps	0	0
748 Kbps	0	1
144 Kbps	1	0
2,320 Kbps	1	1

AAGC Gain	#3	#4	#5
0 dB	0	0	0
3 dB	0	0	1
6 dB	0	1	0
9 dB	0	1	1
12 dB	1	0	0
15 dB	1	0	1
Reserved	1	1	0
Reserved	1	1	1

Transmit Level	#2
4-level	0
2-level	1

Operate NLEC	#1
Bypass	0
Enable	1

Figure C-2. DIP Switch Mode Configuration, UIP Software, or TDEBUG Software, Standalone Mode



Operating Mode	**#8**
Standalone Mode	0
UIP Mode	1

Data Rate	**#6**	**#7**
1,168 Kbps	0	0
748 Kbps	0	1
144 Kbps	1	0
2,320 Kbps	1	1

Test Mode	**#3**	**#4**	**#5**
Normal Operation	0	0	0
BP Isolated +3	0	0	1
BP Continuous 4-Level	0	1	0
BP External Analog Loopback	0	1	1
BP Digital Loopback	1	0	0
Channel Unit PRA to PRA Loopback	1	0	1
Framer Line Loopback	1	1	0
Normal Operation–Unframed	1	1	1

Reserved (should be kept in the 0 [down] position)	**#2**

Terminal Type	**#1**
Local (HTU-C)	0
Remote (HTU-R)	1

Appendix D: Definition of DIP_SW Variable

This appendix discusses the ERLE_MODE_BITS definition that controls the ERLE test configuration (see Section 8.5). When the least significant bit in the *dip_sw* byte is set and the SER_COM directive is not defined, the *dip_sw* byte controls the ERLE configuration. The *dip_sw* byte is of type DIP_SW:

```
DIP_SW    APPL_SW_MSPACE    dip_sw;
```

DIP_SW is a union of three types:

```
typedef union
{
BP_U_8BIT port1;
   PORT1_BITS bits;
   ERLE_MODE_BITS erle_bits;
}DIP_SW;
```

The bit fields in PORT1_BITS are as follows:

```
typedef struct
{
       BP_BIT_FIELD pc_control:1;
       BP_BIT_FIELD symbol_rate:2;
       BP_BIT_FIELD testmode:3;
       BP_BIT_FIELD undefined:1;
       BP_BIT_FIELD terminal_type:1;
}PORT1_BITS;
```

The bit fields in ERLE_MODE_BITS are as follows:

```
typedef struct
   {                                  /* Recommended Values */
   BP_BIT_FIELD transmit_state:1;     /* union with pc_control */
                                      /* set to activate ERLE */

   BP_BIT_FIELD symbol_rate:2;        /* set to 0x0 for 1,168 Kbps */
   BP_BIT_FIELD again:3;              /* set to 0x4 for 12dBm */
   BP_BIT_FIELD transmit_level:1;     /* clear for 4 level */
   BP_BIT_FIELD nl_ec:1;              /* clear for bypass */
   } ERLE_MODE_BITS;
```

The ERLE test can be run at any data rate, but results are typically specified at 1,168 Kbps. Using this data rate and the recommended values for the other parameters (see the comment section in the typedef definition above) gives a value of dip_sw.port1 = 0x21.

Appendix E: Acronyms and Abbreviations

2B1Q	2 Binary to 1 Quaternary
BER	Bit Error Rate
ASM	Activation State Machine
API	Application Program Interface
AERLE	Analog ERLE
AAGC	Analog Automatic Gain Control
DEC	Digital Echo Canceler
DSL	Digital Subscriber Line
DSP	Digital Signal Processing
EC	Echo Canceler
EPROM	Erasable Programmable Read-Only Memory
ERLE	Echo Return Loss Enhancement
FELM	Far-End Level Meter
HDSL	High-rate Digital Subscriber Line
HTU-C	HDSL Terminal Unit - Central Office
HTU-R	HDSL Terminal Unit - Remote
IDE	Integrated Development Environment
Kb	Kilobits
KB	KiloBytes
Kbps	Kilobits per second
KBps	KiloBytes per second
LEC	Linear Echo Canceler
LOS	Loss Of Signal
LOST	Loss Of Signal Time-out
LSB	Least-Significant Bit
MSB	Most-Significant Bit
NLEC	NonLinear Echo Canceler
NLM	Noise Level Meter
NMR	Noise Margin Reading
NOPs	No OPs
RTOS	Real-Time Operating System
SDSL	Symmetrical Digital Subscriber Line
SLM	Signal Level Meter
SNR	Signal-to-Noise Ratio
T-Act	Activation Timer
UART	Universal Asynchronous Receiver/Transmitter
UIP	User Interface Program

Appendix F: References

References:

1. *BtHDSL-EVS User Interface Program (UIP) User's Guide*, October 1996. (UGHDSL_1A)
2. *RS8973 Single-Chip SDSL/HDSL Transceiver Data Sheet*, June 15, 1999. (N8973DSD)
3. *RS8973 Evaluation Module (EVM) Hardware User's Guide*, August 27, 1999. (N8973UGHC)
4. *RS8953B Application and Channel Unit Software Developer's Guide*, May 2000. (100418C).

Appendix G: Revision History

Revision	Date	Description
Rev. A	July 28, 1999	Unedited release
Rev. B	October 4, 1999	Initial release
Rev. C	May 16, 2000	Created Sections 6.4.1 and 6.4.2. Changed minimum response value to –16 in NMR command (Section A.2.4). Changed “Transmit ON SLM” values in Table 8-7.

CONEXANT™
What's next in communications technologies

Further Information
literature@conexant.com
(800) 854-8099 (North America)
(949) 483-6996 (International)
Printed in USA

World Headquarters
Conexant Systems, Inc.
4311 Jamboree Road
Newport Beach, CA
92660-3007
Phone: (949) 483-4600
Fax 1: (949) 483-4078
Fax 2: (949) 483-4391

Americas

U.S. Northwest/
Pacific Northwest – Santa Clara
Phone: (408) 249-9696
Fax: (408) 249-7113

U.S. Southwest – Los Angeles
Phone: (805) 376-0559
Fax: (805) 376-8180

U.S. Southwest – Orange County
Phone: (949) 483-9119
Fax: (949) 483-9090

U.S. Southwest – San Diego
Phone: (858) 713-3374
Fax: (858) 713-4001

U.S. North Central – Illinois
Phone: (630) 773-3454
Fax: (630) 773-3907

U.S. South Central – Texas
Phone: (972) 733-0723
Fax: (972) 407-0639

U.S. Northeast – Massachusetts
Phone: (978) 367-3200
Fax: (978) 256-6868

U.S. Southeast – North Carolina
Phone: (919) 858-9110
Fax: (919) 858-8669

U.S. Southeast – Florida/
South America
Phone: (727) 799-8406
Fax: (727) 799-8306

U.S. Mid-Atlantic – Pennsylvania
Phone: (215) 244-6784
Fax: (215) 244-9292

Canada – Ontario
Phone: (613) 271-2358
Fax: (613) 271-2359

Europe

Europe Central – Germany
Phone: +49 89 829-1320
Fax: +49 89 834-2734

Europe North – England
Phone: +44 1344 486444
Fax: +44 1344 486555

Europe – Israel/Greece
Phone: +972 9 9524000
Fax: +972 9 9573732

Europe South – France
Phone: +33 1 41 44 36 51
Fax: +33 1 41 44 36 90

Europe Mediterranean – Italy
Phone: +39 02 93179911
Fax: +39 02 93179913

Europe – Sweden
Phone: +46 (0) 8 5091 4319
Fax: +46 (0) 8 590 041 10

Europe – Finland
Phone: +358 (0) 9 85 666 435
Fax: +358 (0) 9 85 666 220

Asia – Pacific

Taiwan
Phone: (886-2) 2-720-0282
Fax: (886-2) 2-757-6760

Australia
Phone: (61-2) 9869 4088
Fax: (61-2) 9869 4077

China – Central
Phone: 86-21-6361-2515
Fax: 86-21-6361-2516

China – South
Phone: (852) 2 827-0181
Fax: (852) 2 827-6488

China – South (Satellite)
Phone: (86) 755-5182495

China – North
Phone: (86-10) 8529-9777
Fax: (86-10) 8529-9778

India
Phone: (91-11) 692-4789
Fax: (91-11) 692-4712

Korea
Phone: (82-2) 565-2880
Fax: (82-2) 565-1440

Korea (Satellite)
Phone: (82-53) 745-2880
Fax: (82-53) 745-1440

Singapore
Phone: (65) 737 7355
Fax: (65) 737 9077

Japan
Phone: (81-3) 5371 1520
Fax: (81-3) 5371 1501

www.conexant.com