ipinfusion®

ZebOS®
Advanced Routing Suite
Version 5.4

ZebOS Architecture and Developer Guide

June, 2003

Document Number: 0150301

IP Infusion Inc.
111 W. St. John Street, Suite 910
San Jose, CA 95113

(408) 794-1500 - main
(408) 278-0521 - fax

For support, questions, or comments via E-mail, contact:
support@ipinfusion.com

Table of Contents

Preface

Who should use this Guide

This guide is for routing engineers and other networking professionals who need an overview of the functions and features of the ZebOS® ARS. For details of each protocol, the reader should refer to the developer guides for each of the individual protocols.

This guide describes the ZebOS architecture and illustrates much of its internal operation. It itemizes the list of IETF RFCs that ZebOS fully supports, and it reviews many of the external and internal API functions that facilitate ZebOS module communications and user customization.

IP Infusion Technical Library

Installation and Configuration Guide

Use the *ZebOS Installation and Configuration Guide* to review the hardware and software requirements and to compile and install the ZebOS software. The Installation and Configuration Guide outlines steps for basic configuration of the ZebOS and other routing protocol daemons.

Release Notes

With each new release of ZebOS software, IP Infusion sends Release Notes that describe all the new enhancements and features, the bugs fixed and any new bugs found in the software.

Command References

These manuals describe the Command Line Interface command syntax, describe each of the commands and parameters, and provide usage examples for each command.

- RIP
- OSPF
- BGP
- VRRP
- RMM (Redundancy Management Module)
- MPLS
- RSVP
- LDP
- VR
- ZebOS NSM
- IS-IS (Intermediate System to Intermediate System)
- PIM-SM (Protocol Independant Multicasting Simplex Mode)
- *MPLS Layer-3 VPN User Guide* describes the L-3 VPN-specific APIs.

Developer Guides

These manuals describe the detailed architecture of all the protocols and their interfaces with the ZebOS daemons.

- *ZebOS* (this manual) describes the overall architecture of the ZebOS software.
- *ZebOS NSM* describes the Network Services Module (NSM) interfaces to the other protocol modules.

- *BGP* describes the interfaces to VPN, graceful restart, as well as the API and function calls. Contains detailed descriptions of the SNMP and CLI APIs.
- *OSPF* describes the OSPF-specific SNMP and CLI APIs.
- *RIP* describes the RIP-specific SMNP and CLI APIs.
- *VRRP* describes the VRRP-specific API and function calls.
- *MPLS* describes all the MPLS functionality, data forwarder APIs, and other function calls.
- *Virtual Routing* describes the virtual routing function.
- *ISIS* describes the ISIS-specific SNMP and CLI APIs.
- *RMM* describes the RMM-specific API.
- *VRRP* describes the VRRP-specific APIs.
- *RSVP* describes the RSVP-specific APIs.
- *PIM-SM* describes the *PIM-SM*-specific APIs.
- *MPLS Layer-2 Virtual Circuit* describes the L2 VC-specific APIs.
- *OSPF* describes the OSPF-specific SNMP and CLI APIs.
- *PAL API Reference* describes each API in the PAL.
- *PAL Migration Guide* describes the differences among the various PAL modules.
- *NPapi Toolkit* describes the Network Processor Application Programming Interface.

CHAPTER 1 ZebOS® Advanced Routing Suite

Advanced Software for Next Generation IP Routing

IP Infusion's ZebOS® Advanced Routing Suite (ARS) is one of the most widely used, advanced Internet Protocol (IP) routing and multiprotocol label switching (MPLS) software suites available on the market. ZebOS has a unique modular, scalable architecture that allows OEMs to quickly integrate IP routing and MPLS switching protocols into their platforms. In addition, ZebOS is the first routing and switching package that supports IPv4, IPv6 and MPLS protocols. ZebOS contains a number of features that ensure that device and system developers can quickly bring IP routing software to market.

In development since 1996, ZebOS ARS is currently in use at thousands of sites worldwide and is one of the most popular routing protocol packages on the market. ZebOS ARS, which has been extensively tested both at customer sites and at interoperability labs, is a robust, high-performance software package that meets the demands of today's networks, including carrier-class environments. ZebOS provides all the tools that developers and integrators need to quickly integrate ZebOS ARS into their platforms, including user and developer technical information as well as technical support.

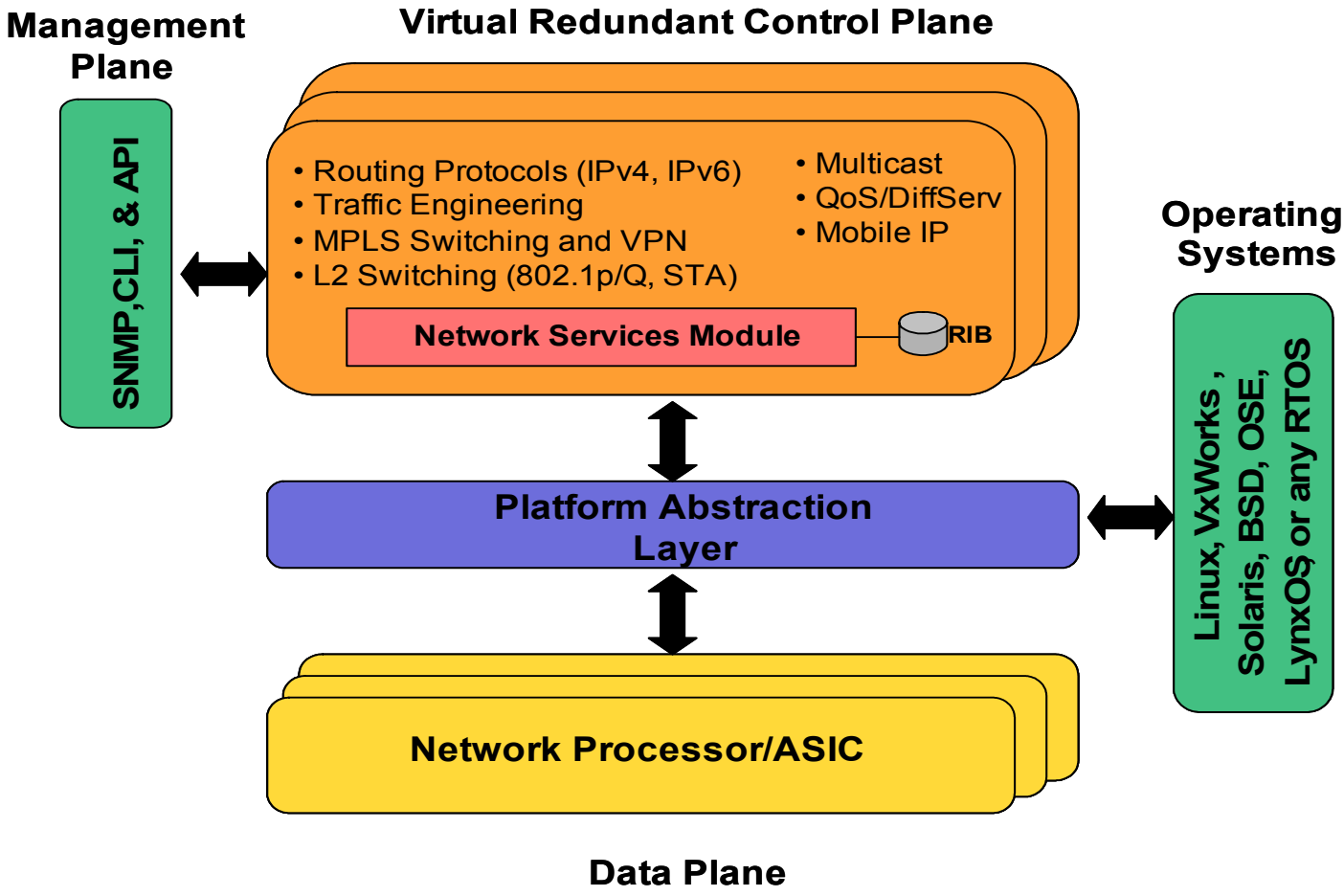
ZebOS ARS delivers solutions that core routing and switching vendors, appliance vendors, as well as network processor and ASIC developers require for quick time to market. This software is ideal for provider edge and wide area network (WAN) aggregation devices, with its integrated MPLS and IP routing solutions. Consequently, ZebOS ARS assists platform developers in rapidly bringing IP routing and MPLS software to market without the cost and time associated with developing and testing routing protocols.

Each supported protocol runs in a separate daemon, built from separate source code files. This gives ZebOS ARS the advantage of fine-grained modularity. Updates, patches, and enhancements can be applied on a module by module basis without disrupting other running protocols.

- IPv4 and IPv6 Routing Protocol Support
- OSPF v2 and v3 Support
- Label Distribution Protocol (LDP) and MPLS Forwarding
- Network Processor Integration
- Industry Standard Command Line Interface (CLI)
- Extensive Logging Capabilities
- MPLS-TE support through RSVP-TE and OSPF/TE and CSPF
- MPLS Layer 3 VPN LDP/RSVP-TE and BGP-VPN Extensions
- PIM-SM
- Redundancy
- RIP v1, v2 and RIPng Support
- BGP-4, BGP-4+ Support
- High-Availability Support
- Simple Network Management Protocol (SNMP) Support and API
- RSVP-TE Support
- MPLS Layer 2 Virtual Circuit
- Virtual Routing
- IS-IS and ISISv6
- Virtual Router Redundancy Protocol
- GMPLS Extension for OSPF

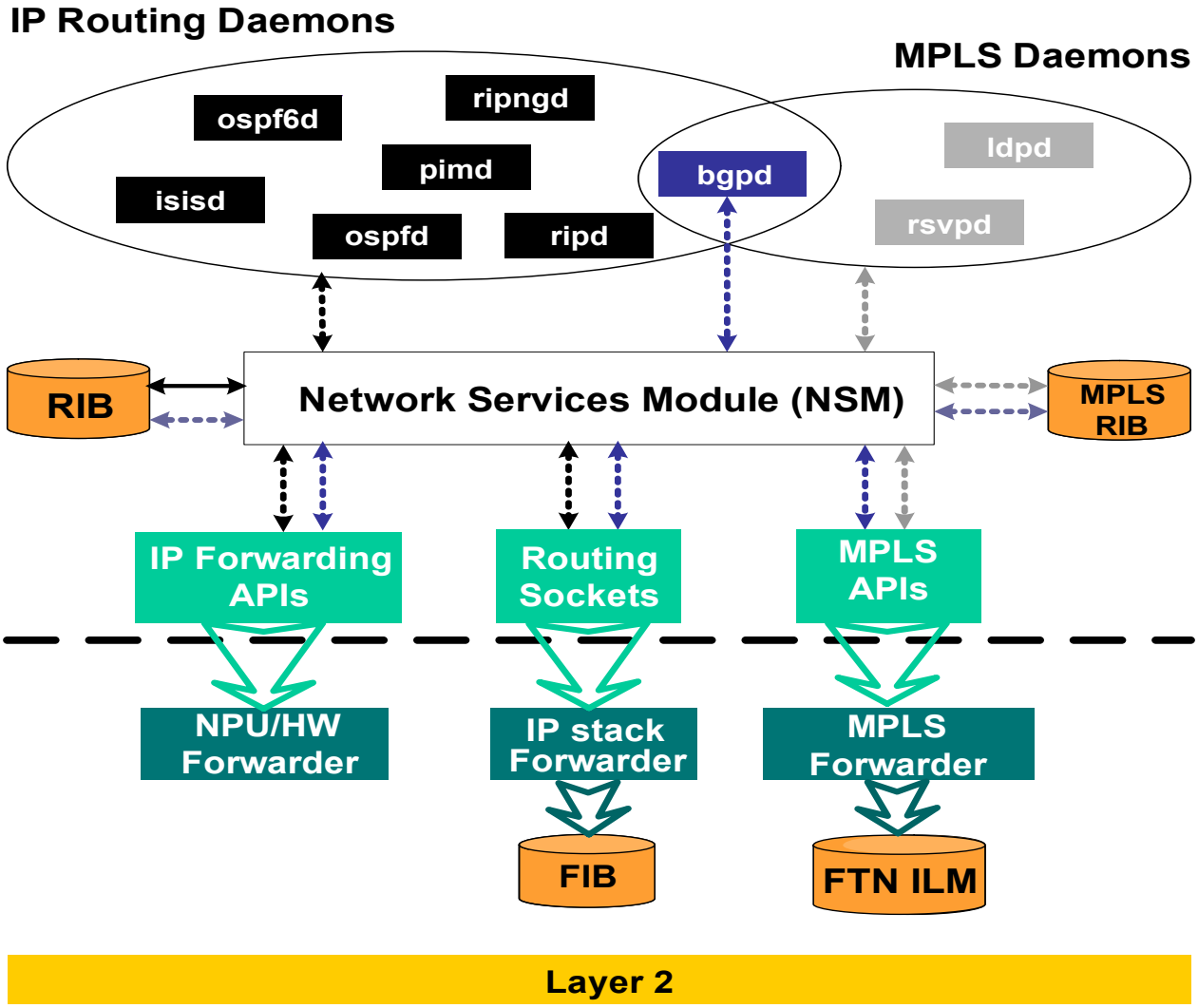
ZebOS Architecture

ZebOS ARS 5.0 utilizes a new scalable, modular, platform-independent architecture for routing and switching software. ZebOS protocols are built on the ZebOS Network Services Module (NSM), which manages the route table and each of the enabled protocols and performs route conversion and redistribution. ZebOS is built with a flexible architecture called PAL (Platform Abstraction Layer) that allows developers to easily expand, adapt or configure it according to their specific routing protocol needs. This well-defined API is available for the ZebOS NSM to interface with the operating system or data plane for routing table updates. Modules are currently available on a number of operating systems. With its well-defined API's and modular design, ZebOS ARS can easily be ported onto custom and real-time operating systems (RTOS) as well as network processor environments.



Because ZebOS ARS supports IPv4, IPv6 and MPLS, it is well positioned not only to support core, edge, and access routing and switching platforms, but also to act as the standard routing protocol for an entire range of future IPv6 enabled devices that include Small Office Home Office (SOHO) gateways; wireless, access, and security devices; and devices that

support Virtual Private Network (VPN) and Voice-over-Internet Protocol (VoIP) technology, and also require Quality of Service (QoS) and bandwidth management.



Routing Protocol Support

ZebOS ARS supports IETF-compliant IPv4 and IPv6 versions of OSPF, BGP, and RIP. The RIP IPv4 Protocol Module supports both RIPv1 and RIPv2. A RIPng Protocol Module is also available for IPv6 platforms. In addition, OSPFv2 and OSPFv3 Protocol Modules are offered for IPv4 and IPv6 support. Traffic engineering (TE) extensions are available for the OSPF and ISIS Protocol Modules. BGP-4+ supports both IPv4 and IPv6 in a single module. Individual spec sheets are available for each protocol module and contain details on supported RFC's, platforms, requirements and features.

IP Infusion is committed to delivering, and has delivered support for a broad range of additional protocols and features. These include multiprotocol label switching (MPLS) signaling protocols, Constraint based-Routing Label Distribution Protocol (CR-LDP), and Resource Reservation Protocol–Traffic Engineering (RSVP–TE), Differentiated Services (DiffServ), Intermediate System to Intermediate System (IS-IS), Virtual Routing, IP Multicast, and an extensive set of Operating System/ Real-time Operating System (OS/RTOS) and network processor ports.

ZebOS ARS consists of many different modules. Each routing protocol is a stand-alone module and is fully integrated into the ZebOS ARS architecture. Each module is an executable task or process. ZebOS is built with a flexible architecture and framework that allow easy enhancements to the existing protocols as well as adding new protocols to the architecture.

ZebOS Network Services Module (NSM) is a common module for all routing protocol modules. NSM manages the route table and each of the enabled protocols, and performs route conversion and redistribution. It also manages the hardware interfaces for all routing protocols. Please see ZebOS NSM Developer Guide for more details. ZebOS NSM provides a well-defined API, called ZebOS protocol, for any routing protocol to leverage the services provided by NSM module via Inter-Process Communication (IPC). Please see ZebOS NSM Developer Guide for more details.

Other routing protocol modules include IETF-compliant IPv4 and IPv6 versions of OSPF, BGP, and RIP. The RIP IPv4 Protocol Module supports both RIPv1 and RIPv2. A RIPng Protocol Module is also available for IPv6 platforms. In addition, OSPFv2 and OSPFv3 Protocol Modules are offered for IPv4 and IPv6 support. Traffic engineering (TE) and Constrained Shortest Path First (CSPF) extensions are available for the OSPF Protocol Modules. BGP-4+ supports both IPv4 and IPv6 in a single module. Label Distribution Protocol (LDP), and Resource Reservation Protocol-Traffic Engineering (RSVP-TE) are also available. Virtual Router Redundancy protocol (VRRP) is a part of NSM module. For more routing protocol specifics, please see the appropriate Developer Guide for more details.

ZebOS ARS implements a set of proprietary system services including threading, finite state machine (FSM), optional memory management, and a command line parser.

MPLS Support

ZebOS ARS supports the LDP and MPLS forwarders to allow IP packets to be switched over an MPLS core network. LDP support for RFC3031, RFC3032, RFC3036, and a number of draft RFCs are provided. The MPLS–LDP Protocol Module runs on top of the ZebOS NSM and uses ZebOS ARS services to obtain routing information. The ZebOS MPLS Forwarder Module is tightly integrated into ZebOS ARS. It works in conjunction with the routing protocol modules to create label-switched paths used when forwarding packets.

The MPLS Forwarder Module also has a MPLS forwarder capability for Ethernet. An MPLS specsheet that contains more details on features, platforms, and requirements is available.

MPLS-VPN Solutions

Most of the current VPN infrastructures are designed using ATM or Frame Relay networks. ZebOS ARS provides a full Layer 3 MPLS-VPN solution by tightly integrating BGP-VPN extensions with our MPLS–LDP Forwarder Module. This MPLS-VPN solution provides address space and routing separation through the use of per VPN routing tables (VRF) and MPLS switching in the core. This provides the security needed by VPN service providers, while at the same time building a scalable infrastructure that can take advantage of IP routing, traffic engineering (TE), and MPLS switching features. Providing VPN services can add significant value to provider edge equipment.

MPLS Layer 2 Virtual Circuit

The MPLS Layer 2 Virtual Circuit (VC) Module is an implementation of draft-martini-l2circuit-trans-mpls-08.txt and draft-martini-l2circuit-encap-mpls-04.txt and is an extension to the existing ZebOS LDP Module. MPLS Layer 2 VC extends a customer LAN across an MPLS network. Ethernet frames from the customer LAN are first encapsulated with a VC label and then sent over an MPLS LSP tunnel to the remote LAN.

Network Processor Integration

The ZebOS NSM supports both an operating system/ real-time operating system (OS/RTOS) abstraction layer as well as a data plane abstraction layer. Our Network Processor API (NPapi™) provides a mechanism to bind the ZebOS control plane functions with popular network processors. IP Infusion is working closely with the Network Processing Forum (NPF) to design and implement a standard API for control to data plane interaction. Although this NPF API is not fully standardized, IP Infusion has implemented a draft standard and is fully committed to tracking and promoting this standard. Please see ZebOS NSM Developer Guide for more details.

Our NPapi currently supports the NPF API submission and will track the standard as it develops. By integrating control and data plane components through its NPapi, IP Infusion helps equipment developers to quickly bring products to market without the difficult integration work that would otherwise be required. IP Infusion is working with a range of popular network processing vendors to ensure that these integrated solutions are readily available and easy to implement.

In addition to NPapi, ZebOS also provides a standard MPLS Forwarding API that is fully integrated with the Linux-based MPLS forwarder module for Ethernet. Please see MPLS Developer Guide for more details.

Management and Logging

ZebOS contains an industry standard CLI to manage and configure both the Network Services Module and all of the protocol modules. The ZebOS CLI follows the industry standard. The ZebOS SNMP Management Information Base (MIB) support allows network management agents, as well as billing and provisioning applications, to pull important data from the ZebOS software and its associated protocol modules for all standard defined MIBs. Furthermore, ZebOS has extensive logging capabilities both to log systems' events and errors. See VTY/CLI Chapter for more details.

SNMP Management Information Base (MIB) support is also available to allow network management agents, as well as billing and provisioning applications, to pull important data from the ZebOS software and its associated protocol modules for all standard defined MIBs. The built-in, pre-ported SNMP agent is SMUX. See Simple Network Management Protocol chapter for more details.

In addition to SNMP and CLI, ZebOS provides a set of application programming interfaces (API) that implement the Command Line Interfaces and MIBs. Use this API to integrate with any management plane to control the ZebOS Advanced Routing Suite. See SNMP-API and CLI-API manuals for more details.

ZebOS has extensive logging facilities to log system events and errors. See Logging and Debugging Chapter for more details.

Nexthop Updating for RSVP PIM and BGP

These three protocols do periodic lookups for prefixes to check router reachability. With the ZebOS Nexthop lookup feature these protocols avoid this periodic lookup. The protocols during start up notify NSM about the prefixes that they are interested in. NSM notifies the protocols if a better nexthop is available or if a nexthop becomes unavailable. This way the protocols need not spend resource with periodic lookups as NSM is proactive in their maintenance.

Development, Documentation, and Support

ZebOS is written in the portable ANSI C programming language. ZebOS is platform and processor independent, but is available with tools and modules to easily support a variety of platforms. ZebOS ARS is delivered with extensive documentation including: Command Reference Manuals, Developer Guides, Installation and Configuration Guide. ZebOS support is available and includes customer assistance and product updates. IP Infusion's support staff is composed of highly skilled network engineers developing, supporting and operating advanced IP networks. Together, these services provide a package that ensures developers can quickly port, integrate, and test the ZebOS Advanced Routing Suite.

System Requirements

ZebOS can be ported to a variety of platforms as long as they support an ANSI C compiler and include an IPv4 and/or IPv6 TCP/IP stack. The ZebOS MPLS–LDP Switching Module can be used with the MPLS forwarder provided, or it can be integrated with existing MPLS Forwarders. It contains both built-in support for a number of standard and real-time operating systems and features to allow vendors to easily port to any RTOS. ZebOS also supports a number of popular processors, including Intel (Pentium), Motorola (PowerPC), and MIPS.

Key Features and Benefits

Feature	Benefits
High Performance Routing Engine	Designed to support millions of routes for both core and edge routing and switching platforms
Modular & Robust Design	Fault tolerant, independent process implementation; easily extensible & upgradeable
IPv4 and IPv6 support	Supports current and future IP implementations and is ready to support explosive growth in IP devices
MPLS Support	Includes LDP and MPLS forwarders for IP delivery over an MPLS core
MPLS-VPN Support	Integrated BGP-VPN and MPLS solution offers the scalability and security of Layer 3 VPNs for provider edge equipment
MPLS L2 VC	Layer 2 MPLS Virtual Circuit support
Network Processor Integration	Support for control and data plane integration through IP Infusion NPapi
Industry standard CLI	Easy and intuitive command line interface for IT personnel
Platform-Independent Design	Works on a variety of platforms, including Linux, Solaris, BSD, and VxWorks, OSE, and so on
API for OS integration	Ability to easily port product to additional platforms
API for Control to Data Plane Integration	Ability to easily port ZebOS ARS to network processor platforms
SNMP Management Support and API	Ability to interface to network management stations and provisioning and billing systems
Member of Interoperability Lab at University of New Hampshire	Full interoperability testing between ZebOS and major router vendors
Virtual Routing (VR)	Provides mechanism to have several isolated routing functions on one hardware platform
Redundancy	Provides mechanism to have failover from primary to backup router (different physical unit) to maintain high-availability (RMM)
PAL	The Platform Abstraction Layer API for porting to any OS.

The Future of the Internet: IPv6 and IP Infusion

The Internet of today is a legacy system, far too limited to support continued, rapid growth. The original design for the Internet Protocol (IPv4) is over 20 years old. A new addressing scheme, Internet Protocol Version 6 (IPv6) improves Internet quality, scalability and security. Some other benefits of IPv6 are:

- **Expanded addressing**. An increase in the available addresses allows many more devices to connect to the Internet and enable the growth of Internet applications (such as VoIP).
- **Simplified header format**. A new header format improves routing efficiency.
- **Improved extension and option support**. Implementations of header extensions improves the ways in which routers process packets.
- **Flow labeling**. Related packets can be treated as streams, improving reliability.
- **Improved Authentication and Privacy**. Security measures are built into the IPv6 protocol.

IETF Membership

IP Infusion is a member of the IETF and will continue its participation in this and other standards bodies. IPv6 functionality is an integral part of the ZebOS ARS and it fully supports the most recent drafts and specifications (RFCs) from the IETF, of the IPv4 and IPv6 versions of RIP, OSPF, and BGP4.

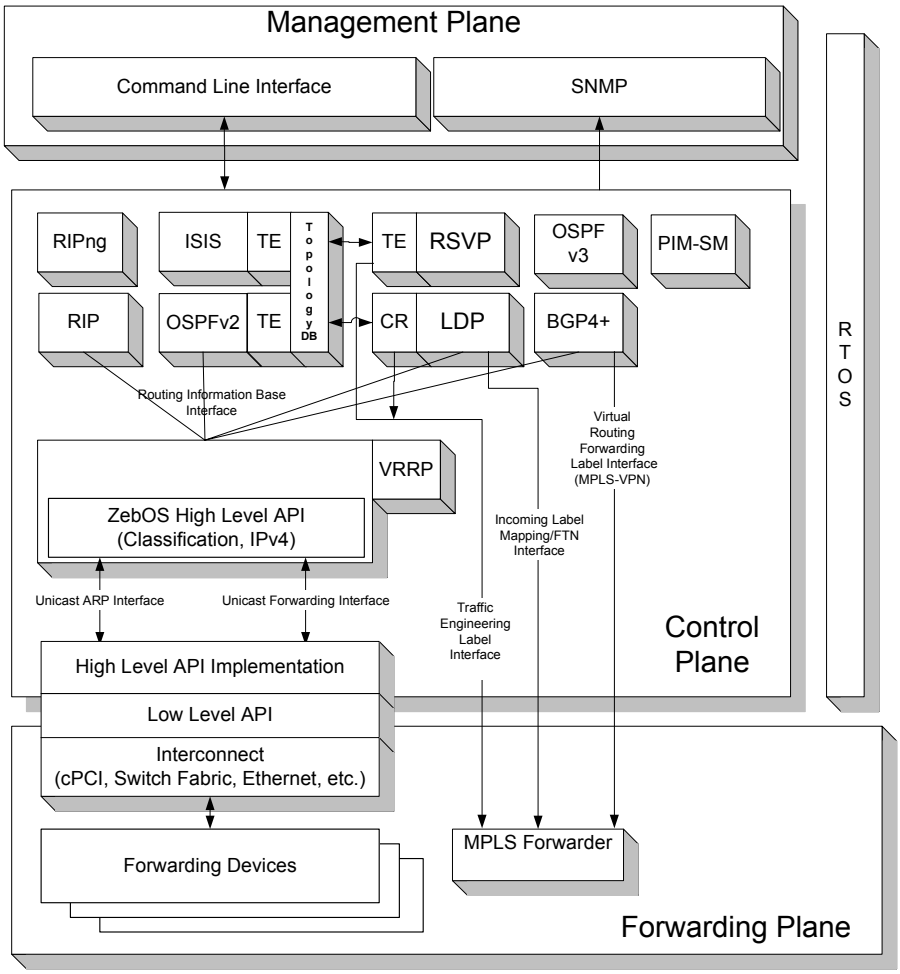
The IPv6 RFCs supported by the ZebOS Advanced Routing Suite are:

- RFC 2080 (RIPng)
- RFC 2740 (OSPF for IPv6)
- RFC 1771 and 2545 (BGP4 and Use of BGP4 Multi-protocol Extensions for IPv6 Inter-Domain Routing)

CHAPTER 2 IP Infusion Product Overview

Overview of Routing Protocols

ZebOS utilizes a scalable, modular, platform-independent architecture for routing and switching software. The ZebOS ARS is a system with three different planes and components.



Routing Information Protocol (RIP)

A distance-vector protocol RIP is an interior gateway protocol (IGP) that uses hop counts as its metrics. The ZebOS ARS RIP module supports RFCs 1058 and 1723; the RIPv2 module supports more fields in the RIP packets and supports security authentication features.

At regular intervals of the routing update timer (a default value of 30 seconds) and at the time of change in the topology, RIP router sends update messages to other routers. The listening routers update their route table with the new route and increase the metric value of the path by one, called a hop count. The router recognizes the IP address advertising router as the next hop and then sends the routing updates to other routers. A maximum allowable hop count is 15. If a router reaches

a metric value of 16 or more (called as infinity), the destination is identified as unreachable. This avoids the indefinite routing loops. The split-horizon and hold-down features are used to avoid propagation incorrect routing information. The route becomes not valid when the route time-out timer expires; it remains in the table until the route-flush timer expires.

Open Shortest Path First (OSPF)

A link-state routing protocol OSPF is an interior gateway protocol (IGP) that uses the *shortest path first* (SPF) Dijsktra algorithm for the Internet and is specified in RFC 1247.

OSPF sends *link-state advertisements* (LSAs) to all other routers within the same hierarchical area. Data on attached interfaces, metrics used, and other variables are included in OSPF LSAs. As OSPF routers accumulate link-state data, they use the SPF algorithm to calculate the shortest path to each node.

An Autonomous System (AS) or Domain is defined as a group of networks with common routing infrastructure. OSPF can work in one AS or receive/send routes from/to different AS systems. Autonomous systems consists of areas. An area is a group of neighboring networks or attached hosts. A router attached to multiple areas with its interfaces is called an Area Border Router (ABR). It creates distinct topological database, a group of LSAs received from all routers in the same area, for each area. All the routers in the same area have identical topological database. OSPF routing traffic is restricted in the area because areas are unknown to each other. The routing information is distributed between areas, area border routers, networks and connected routers by the OSPF backbone.

All backbone OSPF area routers use the same procedures and algorithms to maintain routing information within the backbone that any area router would. The backbone topology is invisible to all routers within an area. The individual area topologies are invisible to the backbone. Sometimes the backbone is not a contiguous area. Virtual links function as if they were direct links and are configured between backbone routers that share a link to a non-backbone area.

AS border routers running OSPF learn about exterior routes through *exterior gateway protocols* (EGPs) such as *Border Gateway Protocol* (BGP).

At boot time, an OSPF router initializes its routing-protocol-specific data structures and tables. When the lower-layer protocols with which it interfaces are functional, it sends the OSPF *Hello protocol* packets to find neighboring routers. A router sends Hello packets as keep-alive packets, informing other routers about its continuing functionality.

Two routers are adjacent when their link state databases are synchronized. *Multi-access networks* have more than two routers. On *multi-access networks,* the hello protocol chooses a *designated router* and a designated backup-router. The designated router generates LSAs for the entire multi-access network, and reduces network traffic and the size of the topological database. The designated router also determines the adjacency of routers and the synchronization of their topological databases. The data on a router's adjacencies or state changes are provided by periodic transmission of an LSA. Failed routers are detected and topology is changed quickly by comparison of adjacencies to link states. Each router calculates a shortest path tree, with itself as a root, from the topological database generated from these LSAs. This shortest path tree creates a routing table.

Border Gateway Protocol (BGP)

Border Gateway Protocol (BGP) is an exterior gateway protocol (EGP) that determines the best path in networks and performs optimal routing between multiple autonomous systems or domains and exchanges routing information with other BGP systems. The RFCs 1771 (BGP4), 1654 (first BGP4 specification), and 1105, 1163, 1267 (older version of BGP) describe BGP and BGP4.

Multiple-peer BGP routers in different autonomous systems or administrative domains on the same physical network support consistent internetwork topology using inter-autonomous system routing. Multiple-peer BGP routers within the same AS support consistent system topology using inter-autonomous system routing. BGP determines the router to serve as the connection point for specific external autonomous system routing services. Multiple-peer BGP routers transport traffic across an autonomous system that does not run BGP using pass-through autonomous systems routing. Here the traffic does not originate or is destined for an autonomous system under consideration, the AS is used only to transport (pass-through) the traffic using some other intra-autonomous system routing protocol.

BGP exchanges information about the list of autonomous system paths with other BGP systems. A connectivity mapping between autonomous systems is created, routing loops are pruned, and other autonomous systems-level policy decisions are taken. Each BGP router maintains a routing table of all feasible and optimal paths to other networks and updates incrementally the routing information received from other peer BGP routers.

The BGP routing metric describes the preference of the path and is assigned to each link by the network administrator. The network administrator assigns this value to a link depending on path criteria such as:

- the number of autonomous systems through which the path passes
- the history of stability
- the line speed
- any delays
- cost per packet

Multi-Protocol Label Switching

MPLS brings the traffic engineering capabilities of ATM to packet-based network. It works by tagging IP packets with labels that specify a route and priority. It combines the scalability and flexibility of routing with performance and traffic management of layer 2 switching. Routing protocols such as OSPF or IS-IS define reachability and the binding/mapping between FEC and next-hop address. MPLS gets routing information from OSPF or IS-IS. No changes are required to routing protocols to support MPLS, MPLS-TE, MPLS QoS, or MPLS-BGP VPNs. It works with any control protocol other than IP and layers on top of any link-layer protocol. At the Network Layer MPLS supports IPv6, IPv4, IPX and AppleTalk. At the Link Layer, MPLS supports Ethernet, Token Ring, FDDI, ATM, Frame Relay, and Point-to-Point Links. It supports almost all transport media (ATM, FR, POS, Ethernet and so on) instead of being tied to a specific layer-2 encapsulation. As it uses IP for its addressing, it uses common routing/signaling protocols (OSPF, IS-IS, RSVP and so on).

In an MPLS (RFC 3031, 3032) network, incoming packets are assigned a label by a label edge router (LER). Packets are forwarded along a label switch path (LSP) where each label switch router (LSR) makes forwarding decisions depending on the contents of the label. At each hop, the LSR strips off the existing label and assigns a new label that tells the next hop how to forward the packet.

Label Switch Paths (LSPs) are configured by network operators to guarantee a certain level of performance, to route around network congestion, or to create IP tunnels for network-based virtual private networks. An LSP can be established that crosses multiple Layer 2 transports such as ATM, Frame Relay or Ethernet. This creates end-to-end circuits, with specific performance characteristics, across any type of transport medium and eliminates the need for overlay networks or Layer-2-only control mechanisms. MPLS evolved from earlier technologies such as Cisco's Tag Switching, IBM's ARIS, and Toshiba's Cell-Switched Router.

RSVP-TE (Label Distribution Protocol)

RSVP-TE extends RSVP and CR-LDP extends LDP to support Label distribution and explicit routing. Traffic engineering selects paths chosen by data traffic to balance the traffic load on the various links, routers, and switches in the network. Traffic engineering is important in networks with multiple parallel or alternate paths.

The label assigned to a packet represents the Forwarding Equivalence Class (FEC) to which that packet is assigned. An FEC is a set of packets that are forwarded in the same manner (for example, over the same path with the same forwarding treatment). An LSP is provisioned using Label Distribution Protocols (LDPs, RFC 3036) such as RSVP-TE or CR-LDP. Either of these protocols establishes a path through an MPLS network and reserves necessary resources to meet pre-defined service requirements for the data path. The LDP specification lets an LSR distribute labels to its LDP peers and enables LSR peers to find each other and establish communication. CR-LDP and RSVP-TE are signaling mechanisms used to support traffic engineering across an MPLS backbone.

MPLS-TE & MPLS VPN

MPLS TE makes use of the following concepts:

- Constrained shortest path first (CSPF) algorithm, a modified version of the well-known SPF algorithm, is used in path calculation.
- RSVP/TE extension or CR-LDP is used to establish the forwarding state along the path and to reserve resources along the path.
- Link state routing protocols with extension (OSPF with Opaque LSAs or TE, IS-IS with Link State Packets TLV (type, length, value)) keep track of topology changes propagation

Since MPLS allows the creation of virtual circuits or tunnels across an IP network, providers use MPLS to provision Virtual Private Network services. There are multiple proposals for using MPLS to provision IP-based VPNs. MPLS-BGP/VPN enables MPLS-VPNs (RFC 2547) via extensions to Border Gateway Protocol (BGP). Here BGP propagates VPN-IPv4 data using the BGP multi-protocol extensions (MP-BGP) for handling these extended addresses. MP-BGP propagates reachability data (VPN-IPv4 addresses) among Edge Label Switch Routers (Provider Edge router). The reachability data for a given VPN are propagated only to other members of that VPN. The BGP multi-protocol extensions identify the valid recipients for VPN routing data. All the members of the VPN learn routes from other members. MP-BGP is implemented using either IPV6 BGP4+, IPv4 multicast (MBGP), or MPLS VPN/BGP (BGP-AD for AutoDiscovery). MPLS can also be used to create IP-VPN's based on the idea of maintaining separate routing tables for various virtual private networks and does not involve BGP.

Resource Reservation Protocol-Traffic Engineering (RSVP-TE)

The RSVP (resource reservation protocol) is an independent protocol that establishes a level of service quality for a data stream or path through a network. Host machines use RSVP to request specific service quality from network for particular data streams. Routers along these data streams use RSVP to deliver quality of service requests to all nodes in a path or data stream, to establish the level of service, and to maintain the quality of service.

This protocol is a signaling protocol that supports explicit routing capability. To do this, a simple EXPLICIT_ROUTE object is incorporated into RSVP PATH messages. The object encapsulates a sequence of hops which constitutes the explicitly routed path. Using this object, the paths taken by the label-switched RSVP-MPLS flows can be predetermined without conventional IP routing. The explicitly routed path can be administratively specified or computed based on CSPF and policy requirements and taking the current network state into consideration.

One useful application of explicit routing is Traffic Engineering (TE). Using explicitly routed LSPs, an ingress node can control the path through which traffic flows from itself, through the MPLS network, to the egress node. Explicit routing is therefore useful for the optimization of network resources and an increase in the quality of traffic oriented performance.

PIM-SM

The ZebOS™ Advanced Routing Suite (ARS) Protocol Independent Multicast–Sparse Mode (PIM-SM) Module is a multicast routing protocol module that uses the underlying unicast Routing Information Base (RIB) to find out the best next-hop neighbor to reach the root of the multicast data distribution tree or the Rendezvous Point (RP) or the source. It builds unidirectional-shared trees per group and optionally creates shortest-path trees per source.

IS-IS

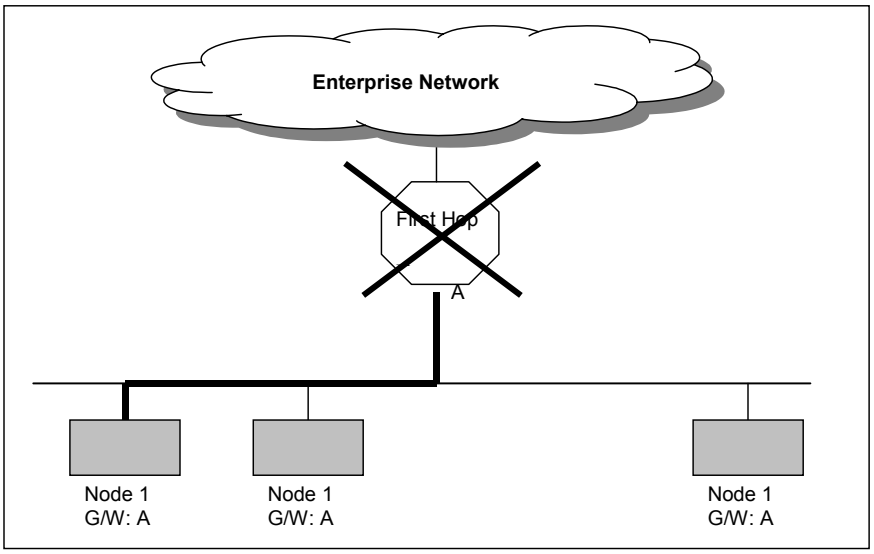
The ZebOS™ Advanced Routing Suite (ARS) Intermediate System-to-Intermediate System (IS-IS) Module provides an IETF-compliant implementation of IS-IS. It is based on link-state technology with two levels of hierarchy. IS-IS forwards the Open Systems Interconnect (OSI) and IP packets unaltered; packets are transmitted directly over the underlying link-layer protocols using the Dijkstra algorithm to find the shortest path to the destination. The ZebOS IS-IS Module communicates

to ZebOS Network Services Module (NSM) to pass routing information to the forwarding plane. The IS-IS Module is fully integrated into the NSM and the other IPv4 protocol modules.

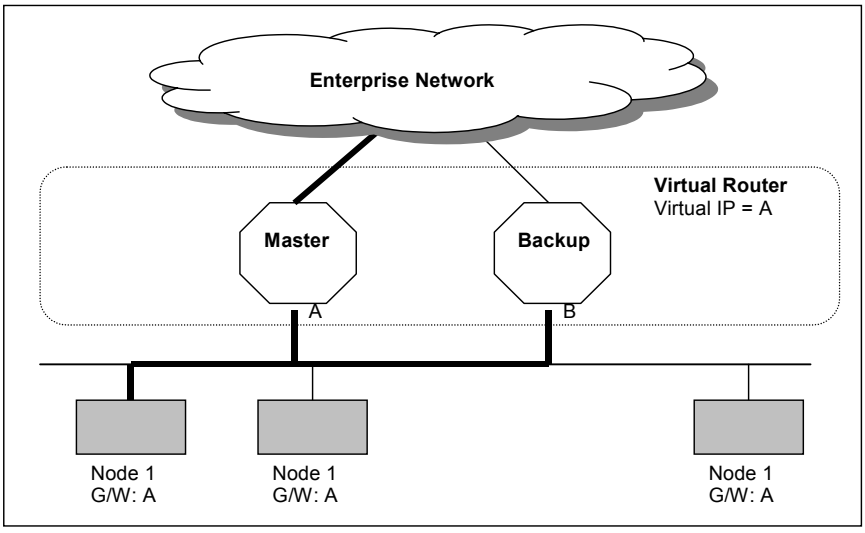
Virtual Router Redundancy Protocol (VRRP)

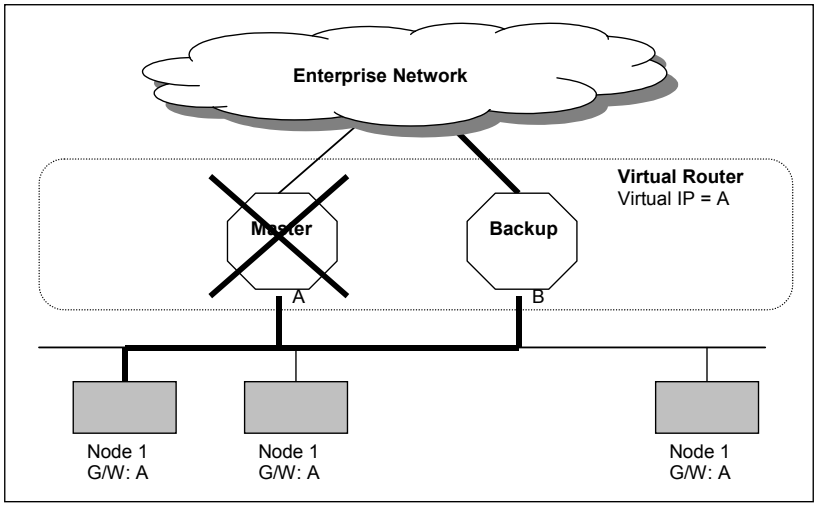
This document provides an architectural overview of VRRP implementation over ZebOS platform.

Typically, end hosts are connected to the enterprise network through a single router (first hop router) that is in the same Local Area Network (LAN) segment. The most popular method of configuration is for the end hosts to statically configure this router as their default gateway. This minimizes configuration and processing overhead. The main problem with this configuration method is that it produces a single point of failure if this first hop router fails.



The Virtual Router Redundancy Protocol attempts to solve this problem by introducing the concept of a virtual router, composed of two or more VRRP routers on the same subnet. The concept of a virtual IP address is also introduced, which is the address that end hosts configure as their default gateway. Only one of the routers (called the Master) forwards packets on behalf of this IP address. In the event that the Master fails, one of the other routers (Backups) assumes forwarding responsibility for it .





At first glance, the above configuration outlined in might not seem very useful, as it doubles the cost and leaves one router idle at all times. This, however, can be avoided by creating two virtual routers and splitting the traffic between them.

Redundancy

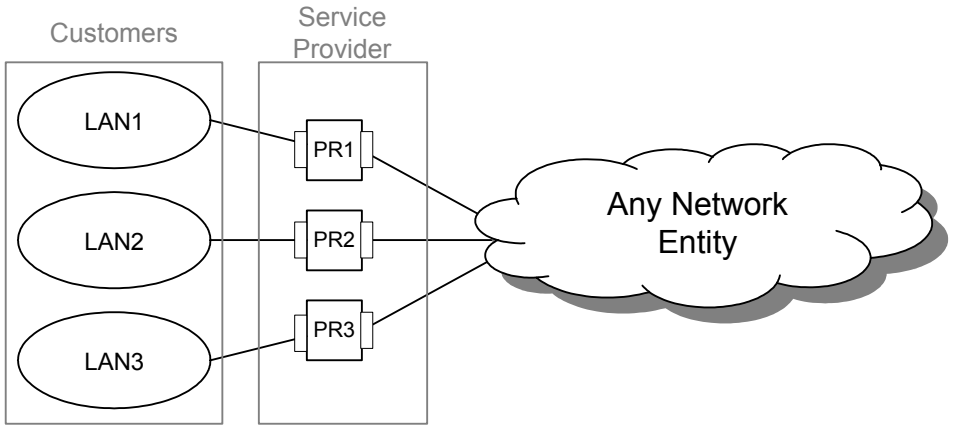
Mission critical applications running on fault tolerant networking equipment such as routers and switches need redundancy and high availability. The Routing Redundancy Module consists of extensions to the Redundancy Management Module (RMM) and the Network Services Module (NSM). The RMM provides a series of APIs that enables any redundant application to function properly within a fault tolerant system. The NSM extensions provide communication between primary and secondary NSM to duplicate the Routing Information Base (RIB). During a failure, the NSM extensions preserve routing/forwarding information and bring the routing protocols back online quickly and with minimal impact to the network.

Virtual Routing (VR)

Physical Routers

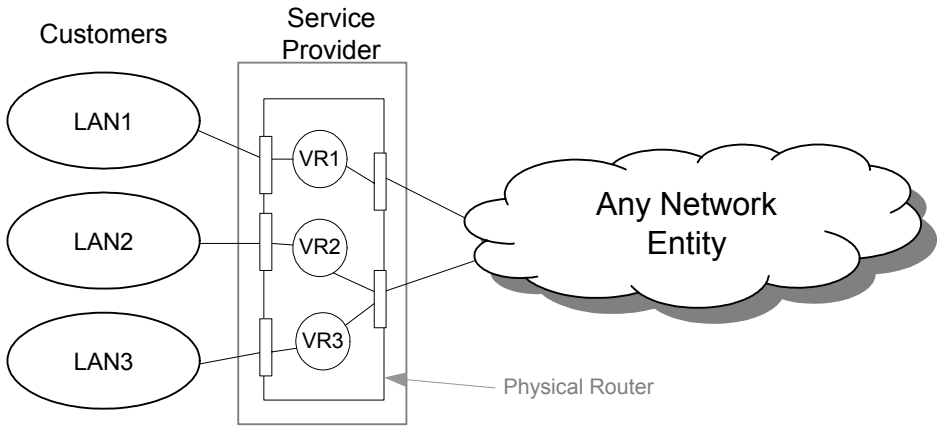
Without virtual routers (VRs), a typical service-provider-to-customer relationship involves one physical router (PR) for services like VPN.

Each router functions as an extension of each customer's LAN.



Virtual Routers

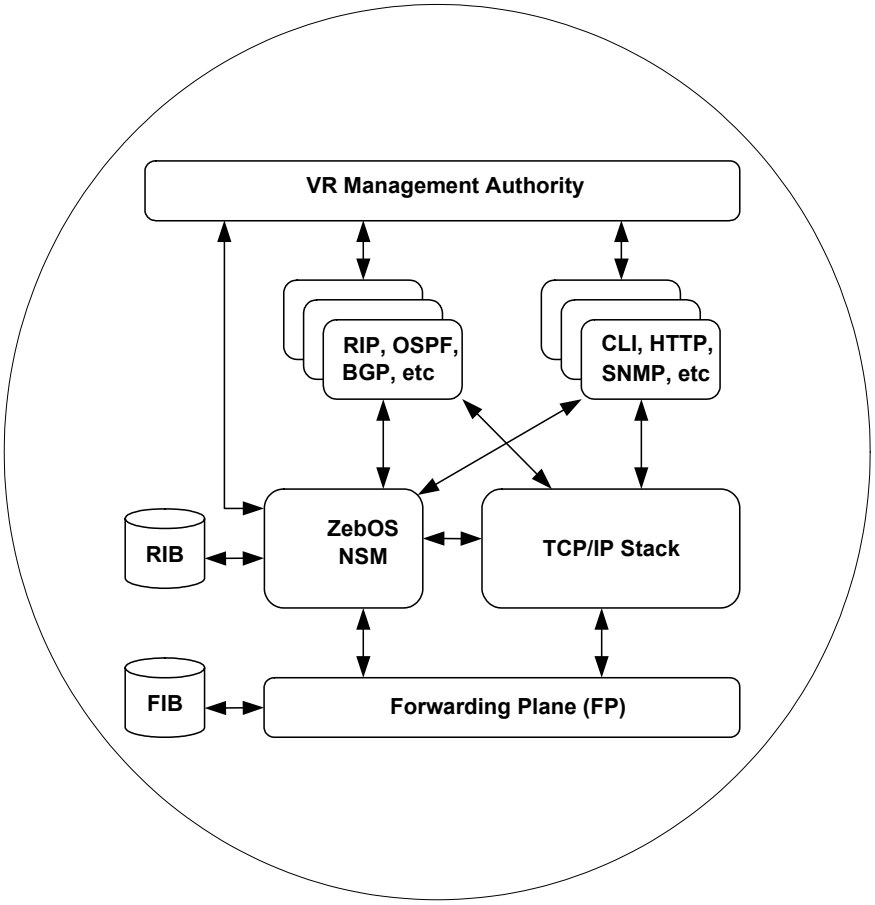
Each VR has its own connection with the customer network, but might share physical resources (for instance, several logical interfaces mapped to a physical one).



Virtual Routers from the inside

A virtual router is a software emulation of a physical router. The many properties of a router are present in a virtual router: addressability, configurability, troubleshooting. All the tools used to configure the software in physical routers are available for virtual routers: command line interface (CLI), scripts, telnet (remote configuration).

Each VR has a management authority, VRMA, that administrators use to configure and maintain the router. Each VR also has a complete copy of the ZebOS ARS, with routing information base (RIB) and forwarding information base (FIB).



ZebOS ARS Supported Standards

RFC or Draft	Descriptions
Draft-bernstein-gmpls-optical-01	Some Comments on GMPLS and Optical Technologies
Draft-chen-bgp-dynamic-cap-02	Dynamic Capability for BGP-4
Draft-chen-bgp-prefix-orf-04	Outbound Router Filter (ORF)
Draft-ietf-diffserv-model-06	A conceptual model for Diffserv routers
Draft-ietf-idr-aspath-orf-02	Outbound Router Filter (ORF)
Draft-ietf-idr-bgp4-17	A Border Gateway Protocol 4 (BGP-4)
Draft-ietf-idr-bgp4-multiprotocol-v2-05	Multiprotocol Extensions for BGP-4
Draft-ietf-idr-bgp-ext-communities-05	BGP Extended Communities Attribute
Draft-ietf-idr-restart-05	BGP Gracful Restart
Draft-ietf-isis-diff-te-01	Extensions to ISIS for support of Diff-Serv-aware MPLS Traffic Engineering
Draft-ietf-isis-ipv6-02	Routing IPv6 with IS-IS
Draft-ietf-isis-traffic-04	IS-IS extensions for Traffic Engineering
Draft-ietf-isis-wg-mib-09	Management Information Base for IS-IS
Draft-ietf-mpls-ldp-mib-08	Definitions of Managed Objects for the Multiprotocol Label Switching, Label Distribution Protocol
Draft-ietf-mpls-rsvp-tunnel-applicability-02	Applicability Statement for Extensions to RSVP for LSP-Tunnels
Draft-ietf-ospf-abr-alt-04	Alternative OSPF ABR Implementations
Draft-ietf-ospf-nssa-update-11	The OSPF NSSA Option
Draft-ietf-pim-sm-bsr-02	Bootstrap Router (BSR) Mechanism for PIM Sparse Mode
Draft-ietf-pim-sm-v2-new-05	Protocol Independent Multicast - Sparse Mode PIM-SM): Protocol Specification (Revised)
Draft-ietf-ppvpn-rfc2547bis-02	BGP/MPLS VPNs
Draft-ietf-vrrp-spec-v2-06	Virtual Router Redundancy Protocol
Draft-katz-yeung-ospf-traffic-07	TE extensions to OSPF
Draft-martini-l2circuit-encap-mpls-04	Encapsulation Methods for transport of Layer 2 Frames over IP and MPLS Networks
Draft-martini-l2circuit-trans-mpls-10	Transport of Layer 2 Frames over MPLS
Draft-ramachandra-bgp-ext-communities-10	BGP Extended Communities Attribute
1058	Routing Information Protocol (RIP)
1195	Use of OSI IS-IS for Routing in TCP/IP and Dual Environments
1227	SNMP MUX protocol and MIB
1321	The MD5 Message-Digest Algorithm
1370	Applicability Statement for OSPF
1587	The OSPF NSSA Option
1657	Definitions of Managed Objects for the Fourth Version of the Border Gateway Protocol (BGP-4) using SMIv2
1724	RIP Version 2 MIB Extension
1765	OSPF Database Overflow

RFC or Draft	Descriptions
1771	A Border Gateway Protocol 4 (BGP-4)
1772	Applications of BGP in the Internet
1812	Requirements for IP Version 4 Routers
1850	OSPF Version 2 Management Information Base
1966	BGP Route Reflection An alternative to full mesh IBGP
1997	BGP Communities Attribute
2080	RIPng for IPv6
2082	RIP-2 MD5 Authentication
2205	Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification
2210	The Use of RSVP with IETF Integrated Services
2211	Specification of the Controlled-Load Network Element Service
2212	Specification of Guaranteed Quality of Service
2236	Internet Group Management Protocol, Version 2.
2328	OSPF Version 2
2338	Virtual Router Redundancy Protocol
2362	Protocol Independent Multicast-Sparse Mode (PIM-SM):
2370	The OSPF Opaque LSA Option
2439	BGP Route Flap Damping
2453	RIP Version 2
2545	Use of BGP-4 Multiprotocol Extensions for IPv6 Inter-Domain Routing
2547	BGP/MPLS VPNs
2740	OSPF for IPv6
2763	Dynamic Hostname Exchange Mechanism for IS-IS
2796	BGP Route Reflection - An Alternative to Full Mesh IBGP
2842	Capabilities Advertisement with BGP-4
2858	Multiprotocol Extensions for BGP-4
2918	Route Refresh Capability for BGP-4
2934	Protocol Independent Multicast MIB for IPv4
2966	Domain-wide Prefix Distribution with Two-Level IS-IS
2973	IS-IS Mesh Groups
3031	Multiprotocol Label Switching Architecture
3032	MPLS label stack encoding
3032	MPLS Label Stack Encoding
3036	LDP Specification
3065	Autonomous System Confederations for BGP
3107	Carrying Label Information in BGP-4
3209	RSVP-TE: Extensions to RSVP for LSP Tunnels
3212	Constraint-Based LSP Setup using LDP
3270	MPLS Diffserv Extension

CHAPTER 3 Source Code Structure Overview

Source Code Summary

In ZebOS ARS, each protocol daemon has its own module and its own source code consists of four logical parts:

- **Generic C source** code: Compile this code with any ANSI-compliant C compiler. The standard environment includes make, bash, and other Unix-related utilities. Source code can be recompiled on a Windows NT platform using the CYGWIN windows environment from `http://www.redhat.com/` .
- **OS-specific** code: encapsulates system specific calls. These calls are related to task creation, message passing, memory allocation, and serial TTY communication.
- **Network-specific** code: addresses the issues related to accessing network routing tables, UDP access for RIP, IP access for OSPF, and TCP access for BGP and network management.
- **Data storage-specific** code: consists of the hooks for persistent data storage for configuration load/save operations, and log and trace information.

\ZebOS\ Subdirectories

The ZebOS main directory contains subdirectories for the zebos daemon and for each routing protocol daemon: ripd, ospfd, bgpd, ripngd, ospf6d, ldpd, and so on. The ZebOS main directory also contains two PAL-related directories: `/pal` and `/platform`.The principle ZebOS subdirectories are:

Directory Name	Description
bgpd	contains the BGP module files.
isisd	contains the ISIS module files.
ldpd	contains the LDP module files.
lib	contains the library function files.
mpls	contains the MPLS module files.
npf	contains the Network Processor module files.
nsm	contains the NSM module files. This is the former zebos directory.
ospf6d	contains the OSPFv3 module files.
ospfd	contains the OSPFv2 module files.
pal	contains subdirectories named by platform that contain the implementation of the API functions needed by ZebOS to run on that particular platform. Each platform type is detailed in subsequent chapters.
pimd	contains the PIM-SM module files.
platform	contains subdirectories named by platform that contain code to start daemons or initialize threads, and other platform specifics of startup, shutdown and signaling.
ripd	contains the RIP module files.
ripngd	contains the RIP next generation module files.
rmm	contains the RMM module files.

Directory Name	Description
rsvpd	contains the RSVP module files.
vtysh	contains the VTY shell module files.

/pal

The `pal` directory contains prototypes for all the API calls for mapping ZebOS to platform services depending on which operating system environment licenses included in the license.

- api
- dummy
- freebsd
- linux
- lynxos
- netbsd
- ose
- ose_ipnet
- solaris
- openbsd
- openbsd_ipnet
- vxworks
- vxworks_ipnet

/platform

The `platform` directory contains all the prototypes and functions for starting up and shutting down daemons.

- dummy
- freebsd
- linux
- lynxos
- netbsd
- ose
- ose_ipnet
- solaris
- openbsd
- openbsd_ipnet
- vxworks
- vxworks_ipnet

`Config.h`, `hconfig.h`, and `acconfig.h` contain `#define` flags for the enable prefixes when compiling or omitting specific modules (such as vrrp, ldp, etc.) or specific features (such as NSSA for OSPF). Each enabled flag creates archive files in the `/lib` directory in each individual modules, for example, `libospf.a` in the `ospfd/lib` directory.

The base software contains the `main.c` file. Each of the subdirectories for each module contains the respective `main` files. For example, `/bgpd` contains `bgp_main.c`, `/ldpd` contains `ldp_main.c`, and so on.

The ZebOS/nsm and ZebOS/lib directories in detail

The `ZebOS/nsm` directory contains all the files related to the zebos daemon. The `ZebOS/lib` directory contains all the files for the library. In the ZebOS framework the library is used to link certain common functions and the zebos daemon to get information about the system (interfaces and routes). The library includes functions and structures for:

- Connecting to the zebos daemon (`zclient.h`)
- Vty management (access method (for example telnet), terminal management and CLI control)
- Command registration
- Access lists (commands and functionality) (`filter.h`)
- Prefix lists (commands and functionality) (`plist.h`)
- Route maps
- Key chains
- Logging
- Linked lists, vectors and hashes
- Memory management, including CLI
- Cooperative multithreading (using select (2))
- MD5 authentication
- Interface structs and functions, including zebos protocol functions (`if.h`)
- Socket management (`sockunion.h` and `sockopt.h`)
- IPv4 and IPv6 route tables
- Internal functions, like serialization support routines for the zebos protocol.

CHAPTER 4 Build System Overview

Compile-time configuration

GNU software licensing

Certain Linux Kernel patches are provided with ZebOS ARS. These are a Linux Kernel Patch for the MPLS Forwarder and a Linux Kernel Patch for the PIM-SM Forwarder. These patches are distributed under the GNU General Public License. They are not part of the ZebOS software. To review the terms and conditions of the GNU General Public License, please go to `www.gnu.org`.

If you contemplate using any open source software that would link with ZebOS code when compiled, please carefully review the license agreement for the open source software first. Software that is licensed under the GNU General Public License may not be linked with ZebOS code because that could require disclosure of the ZebOS source code. That would be a violation of your ZebOS software license agreement. Open source software that is licensed under other types of license agreements may impose similar requirements.

GNU Compiler Versions

To compile PAL, GNU make version 3.79.1 or later is needed. Earlier versions, including 3.79 do not work.

Build Process Files

acconfig.h

This file is input to the autoconf command.

configure

This script file contains statements that determine the system dependent variables and create the makefile. It also parses the `--enable` commands. Use the help parameter `configure --help` to view a list of all the enable options.

makefile

The top-level make file is generic because the platform-specific details are contained in `rules.platform` and `rules.options`.

rules.platform

In each `ZebOS/platform/platformname` directory there is a `rules.platform` file. This file, included by the makefile, contains platform-specific data about:

- the build environment
- installation locations of the final build products
- file suffixes for the build products
- build process tools and commands
- flags for the build process tools

With these and other parameters organized this way, the makefiles are generic, even the top-level one.

rules.options

Some platform directories have a `rules.options` file that defines extra build process flags or module enable flags. This file is included by `rules.platform`.

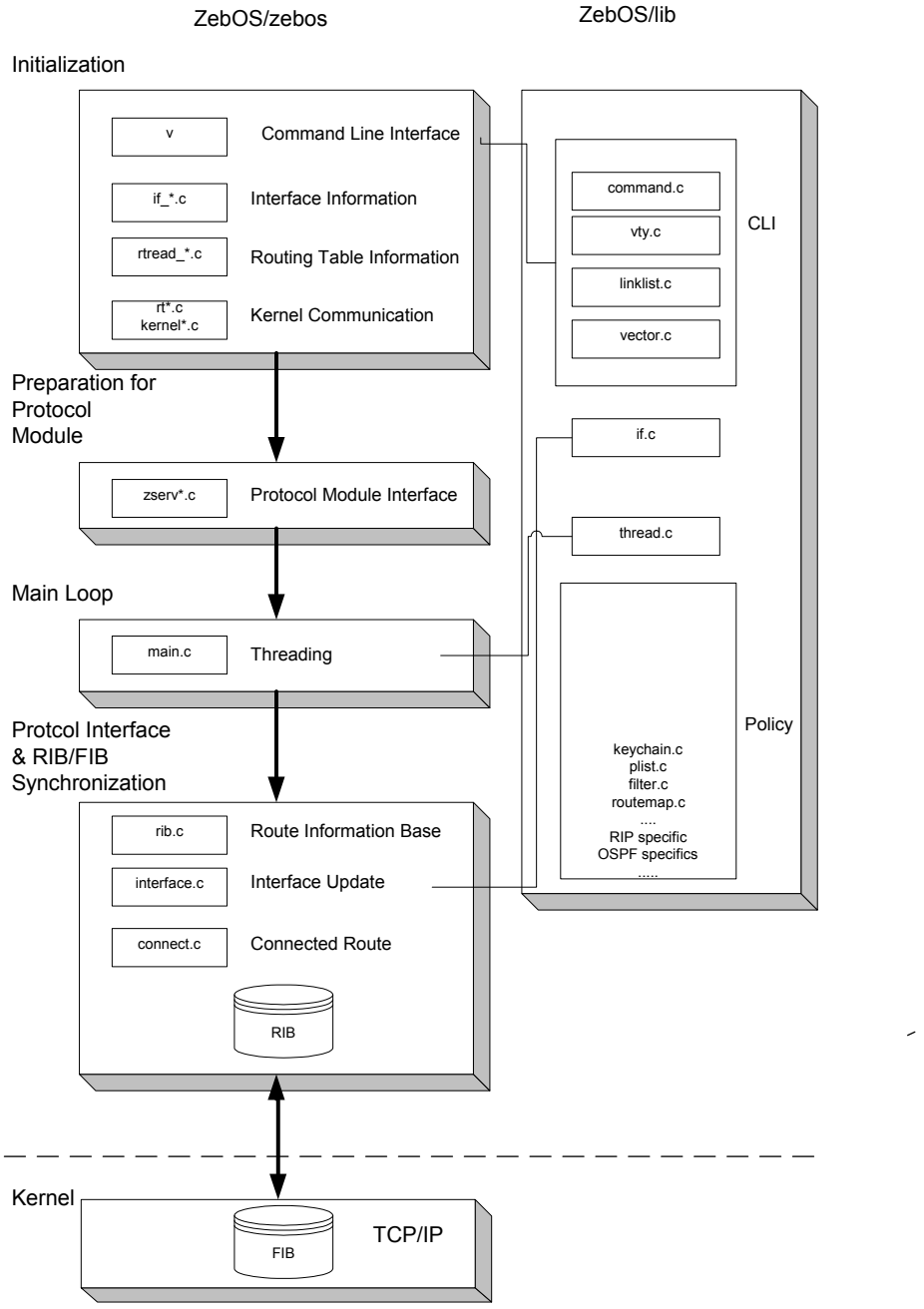
Configuration Options

These options, specified on the command line as part of the `configure` command (`--enable-ipv6`, `--enable-vtysh` and so on) are used by the system in this manner: in the configure script file, there is a list of these options with the defaults specified. This list is merged with the options given with the configure command into the config.h file and the rules.options file. The build process uses the `config.h` file to turn on the several `#define HAVE_` statements.

CHAPTER 5 Common Module Framework

Module Framework

The zebos daemon uses most of the `lib` functions to process its threads, manage lists, manage interfaces, and to manipulate IP addresses. (Example: ZebOS NSM daemon)



Overview of Sample ZebOS NSM

The `main.c` file performs the initialization for vty, zebos, kernel routing sockets, the sorting of the vty commands, and the cleaning up of the self-inserted routes. Then functions within `main.c`:

- read the configuration file `nsm.conf`
- load the configuration parameters
- clear the RIB database
- start the master thread emulator
- fetches and calls multiple threads.

When ZebOS processes start, they read the `nsm.conf` configuration file from storage (disk, memory) with the initial configuration. Through the CLI, users reconfigure each protocol while the protocol is running. The modified configuration can be saved to the configuration file `nsm.conf` on disk, ready for the next start up.

When the routing protocol daemons receive a SIGHUP, they reset themselves, and re-load the current configuration file from disk. Any changes to the configuration not saved in the `nsm.conf` are lost and the process is restarted with the saved settings.

The `nsm_main.c` file has these functions:

- Create VTY socket:
- Print version banner for ZebOS start:
- Use the SIGHUP handler function:
- Use the SIGINT interrupt handler function:
- Use SIGUSR1 handler function:

The zebos process (or task):

- maintains dialogs with all installed routers (through sockets)
- communicates with the operating system for
  - route table information
  - connected interfaces
  - forwarding
  - manages its own route information base (RIB) for centralizing all router and system Route Information.
- reads its own configuration file, `zebos.conf`. This file is in the same sub-directory as zebos itself or in `/usr/local/etc`.
- installs commands
- establishes `vty.c` telnet connections;

The operating system type determines which files will be selected for zebos compilation.

Callbacks are activated as:

- Socket callback Incoming Service Request thread
- Timer callback Any number of re-activation (event or timer) threads
- Command callback An incoming command from the telnet user
- Signal Any of the activated 5 signals.

Overview of Sample ZebOS OSPF

Each protocol could establish a socket connection to ZebOS by way of its own `zebos.c`. Any of the layers could call the Library Functions.

This section introduces how a typical protocol interfaces with the ZebOS NSM services using the OSPF - NSM interface as a model.

ZebOS provides central thread controls to run OSPF timers and processes simultaneously. The central library provides IP-related services, such as IPv4 and IPv6 address manipulation and testing, list management, tree management, route table management, and a variety of other common IP functions used by OSPF, RIP and BGP protocols. ZebOS ARS maintains logging mechanisms and the OSPF interactions with external networks involving other AS connections and RIP and BGP interactions.

Finally, the TCP/IP function interface is provided, including the interface to the routing table. ZebOS supports the data mapping as big endian or little endian, depending on the platform.

The configuration file `ospfd.conf` drives the OSPF operation. Network administrators can edit this file before starting OSPF. Administrators can change the configuration while the daemon is running using the CLI commands; administrators can save the changed configuration with a `write` command to `ospfd.conf`.

OSPF Core Source Files

The build file `configure.in` establishes all the compile-time switches for compiling a specific feature set and environment for zebos and ospfd (and other protocols). The “configure” module then is run to generate `config.h` and `condefs.h`, used for the final compilation and linking.

The `ospf_main.c` file in the ospfd directory has a structure similar to that of the `main.c` file in the main ZebOS directory. It performs module initialization and goes into a loop, processing threads. Each thread completes its functions before the main process begins the next. The main thread is for packet-reading; many timer threads kick-off throughout the OSPF process.

The file `ospfd.h` provides the majority of the data structures involving the Link State Data Base (LSDB), segmented by area.

File `ospfd.c` contains several main functions and provides many of the fundamental features of OSPF.

File `ospf_packet.c` provides direct packet sending and receiving and the initial breakdown of packets by major OSPF type (HELLO, DATA-DESCRIPTOR, LSA-UPDATE, LSA-REQUEST, ACKNOWLEDGE). Code in this file extracts packets from and inserts packets into the database.

File `ospf_snmp.c` provides the centralized SNMP functions for OSPF.

Router Role

Throughout the OSPF process, the particular role of the router is maintained:

- designated router (DR)
- backup designated router (BDR)
- any other router (DRother)
- individual router
- area border router (ABR)
- AS-boundary router (ASBR)
- both ABR and ASBR
- backbone router

- connected in any other arbitrary AREA

Router Connections

A Router can can be connected by:

- broadcast
- point-to-point
- non-broadcast
- virutal.

File `ospf_interface.c` contains the logic for the OSPF module to understand its connections and to map the structures for those connections to other modules.

File `ospf_ism.c` is for the Interface State Machine strictly in accordance with RFC 2319 requirements.

File `ospf_neighbor.c` allows for the definitions of neighbors and adjacencies and provides for the Neighbor State Machine with strict RFC requirements.

File `ospf_lsa.c` can be considered the heart of OSPF, as link state data is exchanged with other systems. This file defines the precise format for all six -types of LSAs as well as the detailed handling for each one.

File `ospf_lsdb.c` contains the logic for installing LSA's into the Data Base.

File `ospf_flood.c` provides for the flooding services of LSAs, depending on the particular role of the Router at the time LSA's are either received, or self-generated. File `ospf_spf.c` can be considered the main reason for OSPF. The calculations for one complete area are performed, with the particular router at the top node; if the router borders several areas, the calculation must be performed for each of those areas.

File `ospf_asbr.c` is the routine for handling all the special conditions when this router is an ASBR. It must advertise its External Connections to the rest of the AS.

File `ospf_ase.c` specifically handles the External Connections incoming from the rest of the AS, and the building of routes on top of the Route Table.

File `ospf_abr.c` specifically handles the processing of LSAs when this router is an ABR. Default LSAs and the Aggregation of LSA packets are handled here.

The above files support various data structures for efficient IP functionality.

File `table.c` describes route table binary tree. An efficient binary tree structure and macros for providing node adds and deletes. It is used about three or four times for Route Table and other intermediate structures.

File `prefix.c` describes IP Data Structures. It contains the IP address structure, in three to four different formats (by use of union).

File `linklist.c` is a set of efficient list functions and macros. It is mainly used for LSAs.

File `stream.c` is a set of intermediate functions and macros for packet streams. It consists of a header and body; typical LSAs are created (copied) as streams then transferred to instantiated memory and the original copy freed.

CHAPTER 6 CLI Overview

Function of the ZebOS CLI

The components of the ZebOS CLI library (in the /lib directory) are: `vty.c`, `command.c`, `linklist.c`, and `vector.c`. These files contain information for the CLIs in zebos, ospfd, ripd, etc. servers/daemons. Basically, vty implementation is a client/server style to handle the command interface from a remote terminal. The client is a telnet session to the server. To fully understand the vty implementation, start with `vty_init()` in the file `ZebOS/lib/vty.c`. The file c`ommand.c` shows how commands are stored and sorted in a linked list (`linklist.c`, vectors) and how the actual functions are invoked for each protocol (all DEFUN functions under each protocol directory).

A telnet session for each module begins in EXEC mode, and progresses to PRIVILEGED mode by use of the enable command. Subsequent modes may involve the CONFIGURATION mode, the ROUTER mode, the INTERFACE mode and others. In addition to the CLI for each module, each module for initial default operation reads a configuration file. This file can be changed by standard editors and saved. The CLI additionally provides the change and write operations for the configuration file. The command `sh run` shows the current configuration file of the module.

The available commands change depending on the configuration mode the user is in. These configuration modes are called nodes in the code. The available nodes are defined by the `enum node_type`:

To enter the debugging state enter the following (abbreviated) commands:

1. `en` for enable to enter privileged mode
2. `de <syntax>` for debug logs to turn on various debug logs
3. `te mo` for terminal monitor to see the debug logs on the monitor

Add a new Command Using CLI

To add CLI commands into ZebOS CLI, the function

```
install_element (struct lib_globals *zg,
                 enum node_type ntype,
                 struct cmd_element *cmd);
```

installs a command with entry `interface_desc_cmd( )` at the INTERFACE_NODE level. If a command is intended to operate at multiple levels, it must be installed into each level.

The entry contains the MACRO definition DEFUN for the Command Header as follows:

- The 1st parameter will be the FUNCTION NAME compiled into the code.
- The 2nd parameter is the INSTALLED NAME entry.
- The 3rd parameter is SYNTAX for the COMMAND.

The remaining parameters are for HELP and LIST command output, corresponding to each parameter of the command (accessed by standard **argc** and **argv**). The code for handling the command follows the Command Header. The return CMD_SUCCESS (or other return codes) finishes the command handler. The variable vty->node maintains the code for the current LEVEL, and all currently defined LEVELs are in the library directory under `command.h`.

Example (interface_desc_cmd used by all protocols)

```
DEFUN (interface_desc,
       interface_desc_cmd,
       "description .LINE",
       "Interface specific description\n"
       "Characters describing this interface\n")
{
  int i;
  struct interface *ifp;
  struct buffer *b;
.
.
  if (argc == 0)
    return CMD_SUCCESS;
.
.
  ifp = vty->index;
  return CMD_SUCCESS;
}
```

Many routines invoke library functions to communicate to the user using the function `install_element()`. The syntax for each user command and its linked handler are specified for each OSPF command.

Access Lists

ZebOS provides a way to define access and prefix lists. When the subsystem with `access_list_init()` is initialized, the user has the following commands available:

```
access-list WORD (deny|permit) (A.B.C.D/M|any)
access-list WORD (deny|permit) A.B.C.D/M (exact-match|)
no access-list WORD (deny|permit) (A.B.C.D/M|any)
no access-list WORD (deny|permit) A.B.C.D/M (exact-match|)
no access-list WORD
access-list WORD remark .LINE
no access-list WORD remark no
access-list WORD remark .LINE
```

If IPv6 support is configured another set of commands for IPv6 access lists is created. The user needs to make sure that `#include filter.h` is inserted into the source code. The following functions are available (from `filter.h`):

Function or Element	Description
void access_list_reset (void);	Removes all access lists
void access_list_add_hook (void (*func)(struct access_list *));	Hook function which is executed when new access_list is added
void access_list_delete_hook (void (*func)(struct access_list *));	Hook function which is executed when new access_list is deleted
struct access_list *access_list_lookup (int, char *);	First parameter is AF_INET or AF_INET6, second is the name of the list, e.g. "101".
enum filter_type access_list_apply (struct access_list *, void *);	Apply access list to object (which should be struct prefix *)."
enum filter_type is {FILTER_DENY, FILTER_PERMIT, FILTER_DYNAMIC}.	Non-existent or empty lists return DENY. Using prefix lists #include "plist.h"
enum prefix_list_type { PREFIX_DENY, PREFIX_PERMIT };	
enum prefix_name_type { PREFIX_TYPE_STRING, PREFIX_TYPE_NUMBER};	/* Prototypes. */
void prefix_list_reset (void);	
void prefix_list_add_hook (void (*func) (void));	
void prefix_list_delete_hook (void (*func) (void));	
struct prefix_list *prefix_list_lookup (int family, char *);	
enum prefix_list_type prefix_list_apply (struct prefix_list *, void *);	The descriptions are similar to the access-list API above.

CHAPTER 7 VTY Overview

Introduction

The ZebOS™ VTY (Virtual Teletype) shell for ZebOS Advanced Routing Suite (ARS) and ZebOS Server Routing Suite (SRS) provides the command line interface (CLI) to the ZebOS daemons. From one telnet session, a system administrator can issue commands that start, stop, reset, display and change the configuration of any accessible daemon. All configuration that can be done directly to a protocol through a dedicated telnet session can be done to multiple protocols through a VTY session.

System status as reported through `show` commands is a consolidated report for all running protocol daemons.

Engineering teams that modify portions of the ZebOS CLI or provide their own CLI, it is important for team members to understand the structure and method of the VTY shell because every command available to each daemon is replicated in the VTY shell.

Features

- Supports industry-standard CLI.
- Supports configuring, monitoring & logging protocols from the CLI.
- Supports word auto-completion & syntax checking.
- Supports ?, help & list.
- Log messages can be logged to a file.
- Easily extendible to add new commands.
- Easily extendible to add new protocol modules.

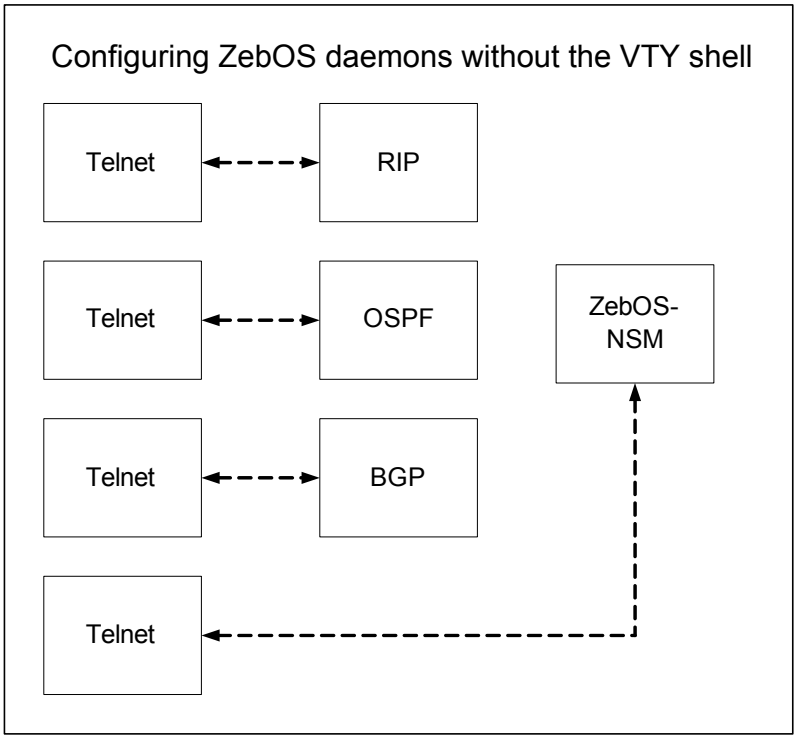
Benefits

- Stable & robust implementation for CLI.
- Fully integrated with ZebOS modules.
- Single point of configure, control & monitoring.
- Platform-independent implementation.

VTY Architectural Concepts

ZebOS without VTY shell

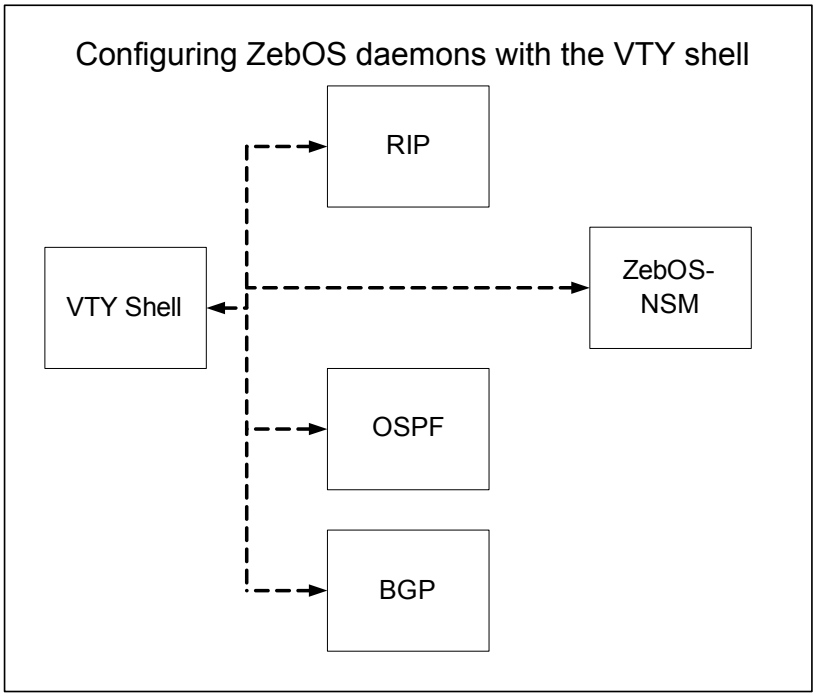
Configuring, maintaining and querying ZebOS daemons without the VTY shell is done with a separate telnet session for each daemon.



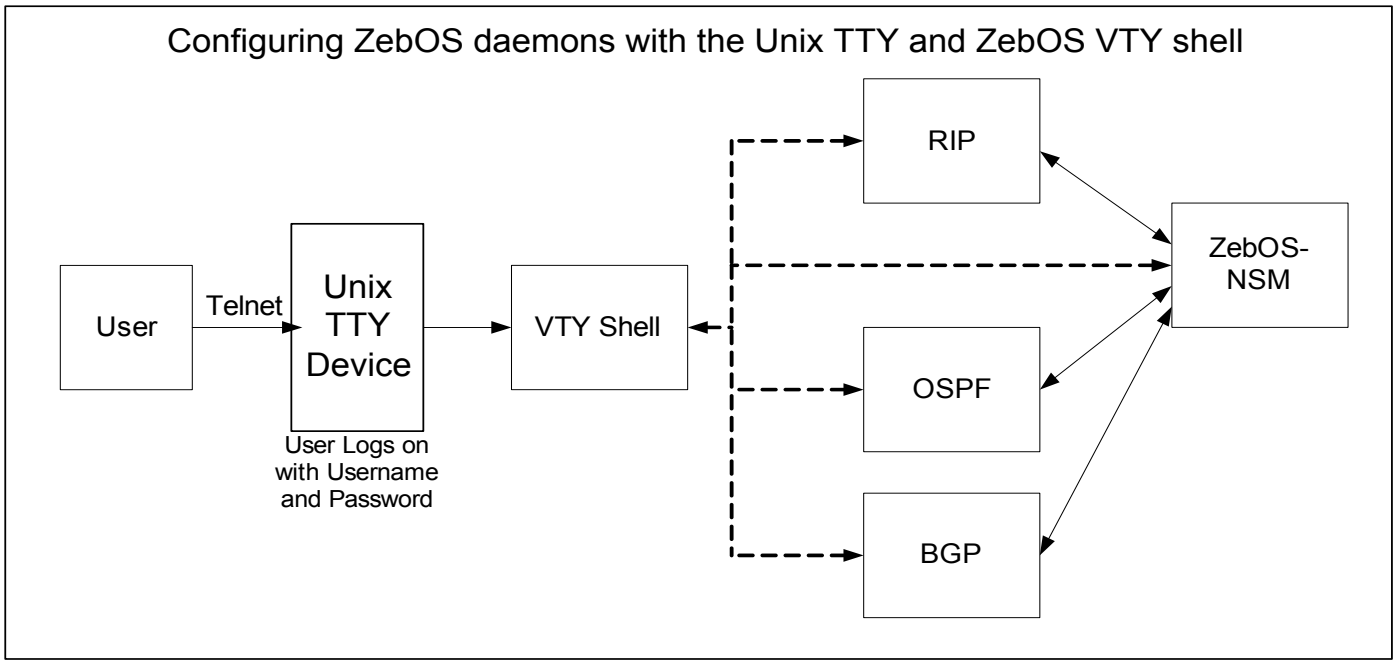
Using VTY shell

ZebOS users with the VTY shell start one telnet session to administer the several daemons in one session window. The telnet session can be on a local machine or on a remote machine.

There are two methods for using the VTY shell: One is by using the VTY shell directly to control the daemons.



The other method is to use a Unix TTY device to enforce user authentication. The user telnets into the TTY device, and logs in with a username and password to the vty shell. From there the user can access the ZebOS daemons..



Unique Command behavior

Some ZebOS CLI commands are not applicable if a user uses the VTY shell to configure the router:

- (no) smux peer
- (no) smux peer password

Note: Routing protocols use the default OID and the default password "ipinfusion"

All commands available to various protocol modules are available to vtysh.

How VTY fits into the ZebOS Build Process

To build vtysh with ZebOS, specify `--enable-vtysh` as an option to `./configure`. This builds vtysh. All the DEFUN/ALIAS from various modules make their way in `ZebOS/platform/platformname/obj/vtysh/vtysh_cmd.c` file.

Building and Customizing VTYsh in the ZebOS ARS Environment

If the user wishes to add/delete some DEFUN/ALIAS from this file depending on his configuration options, he should do the following.

1. # ./configure --enable-.... (all options that user wants)

   ```
   ./configure --enable-vtysh --enable-ipv6 --enable-te --with-libpam --enable-ospf-te
   --enable-mpls --enable-vrrp-linux --enable-mplsvpn --enable-static
   ```

2. cd/ZebOS/platform/platformname

3. # make all or #make vtysh

The `vtysh_cmd.c` file is generated automatically every time vtysh is built depending on the latest configuration options that have been specified. This changes protocol behavior in 2 ways:

- Disables a protocol module from reading "protocol.conf" file at startup.
- User cannot directly telnet to the protocol for configuration/management.

These help to avoid inconsistencies among the configurations of various protocol modules in ZebOS when they are being configured centrally.

Source Code Files

Main Functionality Files

vtysh.c

This file is broken in to several parts:

- VTY main functionality prototypes
  - void vtysh_init_vty ();
  - void vtysh_init_cmd ();
  - void vtysh_connect_all ();
  - void vtysh_readline_init ();
  - void vtysh_user_init ();
  - void vtysh_execute (char *);
  - void vtysh_execute_no_pager (char *);
  - char *vtysh_prompt ();
  - void vtysh_config_write ();
  - int vtysh_config_from_file (struct vty *, FILE *);
  - void vtysh_read_config (char *, char *, char *);

- void vtysh_config_parse (char *);
- void vtysh_config_dump (FILE *);
- void vtysh_config_init ();
- void vtysh_pager_init ();
- void vtysh_config_parse ();
- void vtysh_parse (char *, void (*) (char *));

- The command definition section:

  In this section there are several statement types:

  - DEFUNSH
    Calls the callback in vtysh and sends the command to protocol modules.
  - DEFUN
    Calls the callback in the module in the defined context: the protocol module or vtysh.
  - ALIAS
    Calls the callback for the associated DEFUN statement. Each ALIAS statement is associated with a DEFUN.
  - DEFSH
    Commands of this type are passed to the protocol modules only. Used by vtysh only.
- The node or mode installation section, that defines the prompting strings
- CLI installation section that installs the commands.

vtysh.h

This file contains definitions for the daemons and common macros, macros for the locations of various files and the name of the default configuration file.

vtysh_cmd.c

This dynamically generated file contains all the commands for all the daemons in DEFSH statement and installation commands to install the commands in to the VTY shell.

vtysh_config.c

This file:

- contains functions that parse and execute the commands entered in the VTY shell session.
- takes care of the configuration functions required by vtysh.
- contains a config structure containing the configuration info of the node.
- create, read and write to configuration files.

A config file contains the routing daemon configurations. This information forms the initial command set for a routing daemon as it is starting. Config files are located in:

/usr/local/etc/*.conf

vtysh_main.c

This file

- contains the text displayed by the help command. The help information is called by the main() program in response to the “h” parameter or no parameter.
- calls various functions to initialize vty shell.

vtysh_exec.

- connects to each of the routing daemons through socket connections.

- once the connection is established it parses the commands from the command prompt
- after verifying the command, executes the command if required, passes on commands to protocol daemons and waits for the next command.

vtysh_user.c

This file contains code to log in a user. This file takes care of the authentication. The original version of vtysh uses PAM (plugable authentication module) for this purpose, and authenticates based on the process id that invokes the vtysh. This file makes use of the `geteuid` and `getpasswd` function calls for the purpose.

vtysh_user.h

This file contains a single prototype for the authentication function.

ZebOS/platform/platformname/vtysh.c

This file does signal handling and calls vtysh_main() to do all the work; it contains the main() routine.

libedit

This is the replacement of readline library which acts exactly like readline library. It manages the user input and "command line" editing.

Runtime Files

The file, `ZebOS.conf`, is the optional configuration file for setting up the runtime environment for the VTY shell. Particular features to note in this file are:

- `vtysh` comes up with the following options to read config files.
  - -b, --boot Execute boot startup configuration
  - -f, --file Execute this config file
- If either "`--boot`" or "`--file`" option is specified then `vtysh` reads the config file and passes the configuration information on to all protocol modules.
- If neither option is specified then no configuration file is read.
- Default config file for vtysh is `ZebOS.conf`. At `configure` command time this option, `--prefix=prefixname` contains the name of the directory of the configuration file and is interpreted as `prefixname/etc/`. The default location, if these options are not specified, is `/usr/local/etc/`.

VTY Shell Utility

The VTY shell command extraction utility `extract.pl` is a PERL script that inspects all the ZebOS daemon CLI source files for commands and then builds the `vtysh_cmd.c` file. The extraction utility builds `vtysh_cmd.c` file everytime the vtysh is built.

The `extract.pl` perl script extracts DEFUN and ALIAS statements from files in the various protocol modules and creates the `vtysh_cmd.c` file. If multiple protocol modules have the same command, logic within the script file merge these equally-named commands into one command.

Some commands, such as `show running-config`, need special treatment at command execution time. The script contains logic so that when the command is executed, VTY merges the output from all the daemons into consolidated output.

vtysh Debugging

The vtysh facility is an integrated shell for ZebOS routing engine. The vtysh module allows one Telnet Session to control all the ZebOS routers. It requires the standard Linux/Unix inetd services, and requires the —enable-vtysh" directive to the "configure" program. To configure "vtysh", use the "—enable-vtysh" option in the "configure" command:

```
./configure –host=i686-gnu-linux –enable-vtysh
make
make install
```

To start the "vtysh" daemon:

1. Copy the "vtysh" program to the executable directory
   `cp /…../ZebOS/vtysh/vtysh /usr/local/sbin`
2. Set up a vtysh account using `vipw` to edit document `/etc/passwd`, and include an entry such as `zebos:x:1100:1000:test:/home/<name>:/usr/local/sbin/vtysh`.
3. To locally invoke `vtysh` enter `/usr/local/sbin/vtysh` with an option `-e command` where `-e` specifies the command to be executed in batch mode.

   To remotely invoke "vtysh", install the "inetd" modules. From the remote machine, "telnet <ip_address>"

To generate a starter configuration file, `zebos.conf`, start vtysh, and use the command write memory to generate a copy of `zebos.conf`.

To get information on vtysh, use "man vtysh".

CHAPTER 8 SNMP Overview

Network management typically consists of stations on the network that issue query packets to an IP address for the purpose of gathering status information. IP Infusion protocol daemons respond to these query packets under the requirements defined in RFC 1657, 1724 and 1850. ZebOS requires the use of an intermediary agent using the SMUX protocol specified in RFC 1227. The protocol for the query, set and callback functions are defined as SNMP (RFC 1157) and the Data Descriptions are defined as MIB-II (RFC 1213). ZebOS does not support SNMPv3.

To make SNMP operational within ZebOS, define the `HAVE_SNMP` pre-processor variable; the `enable_snmp` directive performs this operation during compilation. In general, the SMUX agent responds to UDP-161 packets and transmits them to ZebOS over telnet connections using port 199. Each protocol daemon contains its `x_snmp.c` module (where `x` is RIP, OSPF, BGP, or other protocol name) for handling `get`s and `set`s; data flows into and out of the `smux.c` located in `/lib`.

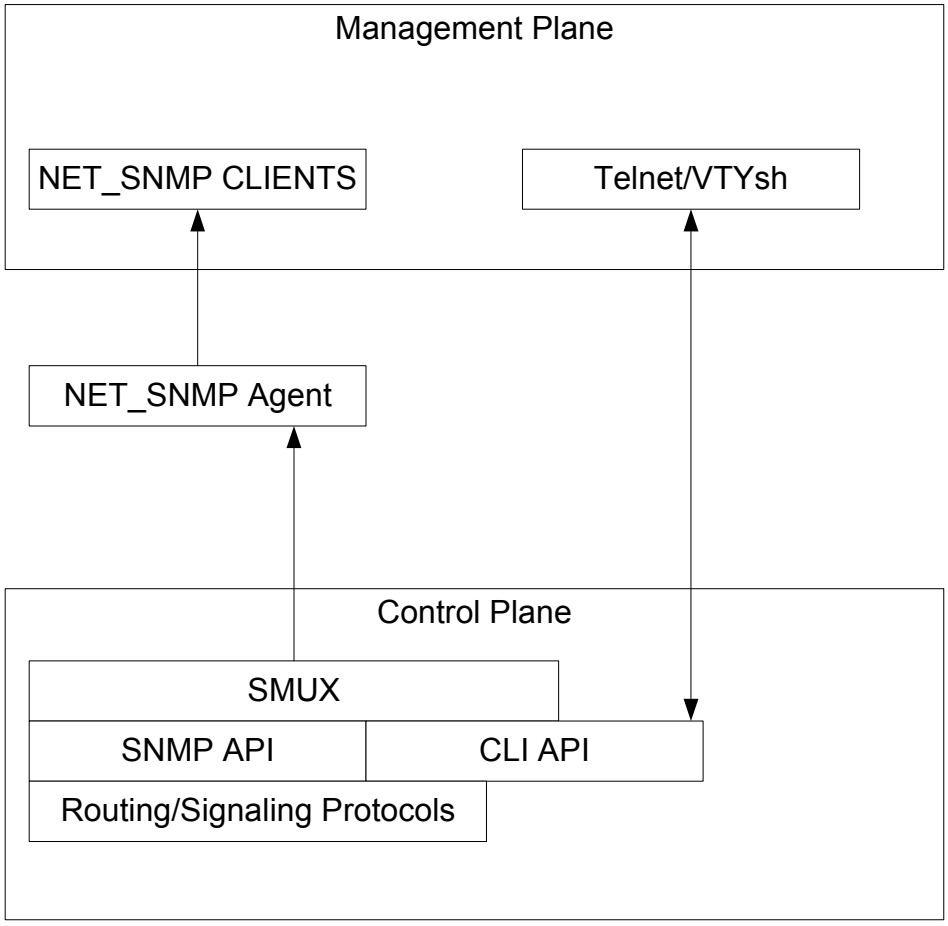
In `/lib/smux.h` `SMUX_PORT_DEFAULT` is defined as 199.

The ASN parser

We

CHAPTER 9 ZebOS CLI and SNMP API Overview

ZebOS offers an SNMP API calls and a Command Line Interface API that together give engineering teams the flexibility to customize these to interfaces. These APIs allow full integration into any management plane.



CHAPTER 10 PAL Overview

Introduction

ZebOS™ ARS advances platform independence with the introduction of the Platform Abstraction Layer (PAL) collections of APIs. For each supported platform, these APIs are fully implemented and use the system services of each platform to isolate ZebOS daemons from operating system details. Developers can concentrate more on feature enhancement and less on porting their changes for multiple OSs. (*platform* is an OS-hardware combination.)

Some PAL equivalents to system components and features:

OS Services	PAL_Services
Types	PAL_Types
Sockets	PAL_Sockets
Memory	PAL_Memory
String	PAL_String
Stdlib	PAL_Stdlib
Configuration/storage	PAL_Config
Interfaces	PAL_Interfaces
Logging	PAL_Log
Route	PAL_Route
Daemonization	PAL_Daemon

In the earlier versions of ZebOS, each ZebOS component had relationships with each OS, management protocol, and processor. For example, differences from one OS to another were handled by the source through #ifdef <OS> statements. With six to eight OSs supported, the code was becoming quite complex.

With PAL, ZebOS supports the same number of OSs and management protocols, and now the complexity is removed from the ZebOS components and pushed to a PAL module for the OS. Each ZebOS component makes one call and the compile time options link in the proper PAL module to fulfill those calls.

Memory Types

New memory management in PAL gives developers more control over memory including the class of requested storage and size of the storage.

Sockets

Sockets services are used to create TCP sockets. PAL sockets handle all traditional socket functionality like creation and deletion, read and write, and synchronization functionality such as select. There are extensions to the socket calls that handle L2 and raw sockets, offering flexibility to those systems that need to handle sockets in a special manner.

String

The String abstraction includes string operations, such as copying, scanning, parsing, and comparing strings and characters. It interacts with the memory abstraction to create or move memory cells, when needed, such as by strdup.

STDLIB

The stdlib abstraction allows developers to take advantage of the most effective stdlib functions on the system. If there is richer or special functionality available for these features, that functionality can be used in preference to the classical function.

Configuration Storage

The PAL abstracts storage of configuration files. This allows platforms without a file system to store configuration information in whichever memory model is available such as linear memory or flash memory.

Logging

Logging facilities are abstracted in such a way that the system can log to whatever facility it has available, be it syslog, file, or the console. These outputs can also be run in parallel, as demonstrated in some of our implementations, where each of several outputs can be set independently to different log trap levels.

Kernel (Interfaces and Routes)

The PAL abstracts control and management of kernel routes and interfaces so ZebOS does not need to be directly aware of how routes are manipulated.

Daemons

Process daemonization on systems is abstracted so that it is automatically handled on systems with this functionality.

Source

The source tree layout puts each protocol in its own directory. There is no platform-specific source in any of the protocol directories. All calls, data types and other components that are operating system or platform specific are moved to the PAL code. All included modules either document other needed included modules or automatically include the necessary modules.

Build

A "dummy" build environment helps to determine if the protocols include any system functions. The dummy environment excludes the system header files. Every protocol that builds with the dummy environment does not include any system-dependent functions. This helps IPI ensure that all of the modules run on any PAL supported environment, even if that environment does not fully support the usual stdlib features, such as file I/O.

The delivered version of ZebOS 5.0 comes with a set of production-quality, fully-tested PAL modules for the most common operating systems. Any porting efforts to non-supported platforms can be modeled on these PAL modules.

In the future, additional platforms can be supported by the PAL. IP Infusion will consider efforts to support other operating systems and assist customers in their porting efforts.

Module Compatibility

Changes, additional features and enhancements to protocols are restricted to the source for that module. Changes made necessary by a platform are restricted to the PAL module for that platform. The PAL API will only change to support new operating system features ensuring backward compatibility for the protocol modules with later versions of the PAL API.

CHAPTER 11 Memory Manager

Introduction

The ZebOS ARS uses dynamic memory. This allows it to operate in constrained environments where little memory is available and in large-scale systems where there is more to track. However, sometimes there are needs beyond allocating as much memory is needed at a given time. Often, certain system conditions change rapidly, and the overhead for memory alloc and free is high on most operating systems. Additionally, the system may want to track memory usage and provide information about how much memory is in use for what purpose, and maybe even statistics about memory allocation, such as the number of times a particular type of memory has been allocated or freed.

Prior versions of the ZebOS ARS provided a memory accounting system to track the active cells and alloc bytes for various purposes. Also, they provided a memory manager feature that, when enabled, would allow different types of memory to be allocated using different methods.

ZebOS ARS moves the memory management function to the Platform Abstraction Layer. In this, it removes the accounting and memory allocation management from the library, that is intended to be platform independent. To replace this functionality, ZebOS provides a sample implementation for a memory manger. In some cases, the platform provider may provide a different memory manager, perhaps with improved abilities on that platform (an example here would be to allocate often used `MTYPE`s (such as route entries) from fast SRAM and to use slower DRAM for less used or less critical `MTYPE`s (such as CLI strings)). Because these rules would vary from platform to platform, and possibly between board revisions, IPI does not supply examples.

This sample implementation combines accounting, statistics, integrity checking, and handling of different allocation methods into a single module, with each feature able to be enabled or disabled at build time, according to the needs of the user and environment.

A memory type feature, the `MTYPE`, provides information to the memory manager about the intended purpose for a particular cell at allocation time. The included memory manager code uses the `MTYPE` to determine how to handle a memory request. Vendor implementations can extend this and use the `MTYPE` to determine not only how to handle memory allocation, but where, or to manage other behavior, as they see fit.

The included memory manager code is written so that the internal logic is platform independent, but it takes into account several aspects of the platform to enhance performance and reliability. As shipped, it is configured to operate on 32-bit protected architectures, but it can be modified through changes to `#define` and `typedef` statements within the module to handle non-protected architectures and 64-bit architectures, and potentially other environments.

The included memory manager code provides support for several allocation methods, and can be extended by the vendor to allow others if desired. As shipped, it supports `HEAP` (pure standard heap allocation), `FSHEAP` (heap allocation using fixed size cells), `PRE` (preallocate only), and `EXPRE` (expandable-preallocate: preallocates cells, but adds more if they are needed). For prealloc types (when expansion is not needed), memory transactions operate in O(1) time. For heap types (or `EXPRE` when expansion is required), the time is based upon the OS heap memory calls.

A configuration file is used to configure (at compile time) allocation behavior for `MTYPES`, as well as to define several parameters for each `MTYPE`.

Because of the complexity in the memory manager, and also because support for each platform is separate to improve portability, there are several files that affect the memory manager:

- `pal_memory.c` file, in the appropriate PAL subdirectory, contains implementation for the memory manager.
- `pal_memory.h` file, also in the appropriate PAL subdirectory, contains definitions to resolve parts of the PAL memory API that are not included in the memory manager implementation onto the API provided by the OS, as well as the platform specific `#define` and `typedef` statements to configure the memory manager.

- `pal_string.c` file, in the appropriate PAL subdirectory, contains implementation for the string functions. This file must not implement the pal_strdup call, as the memory manager needs to do this to maintain integrity.
- `pal_string.h` file, in the appropriate PAL subdirectory, contains definitions to resolve parts of the PAL string API. This file must not resolve the pal_strdup call directly onto the OS supplied function, as the memory manager needs to do this to maintain integrity.
- `pal_memory.def` file, in the PAL API subdirectory, contains the actual function prototypes and descriptions for the PAL memory API. These functions are implemented by the memory manager or mapped onto OS functions as appropriate.
- `pal_memory_desc.def` file, also in the PAL API subdirectory, contains descriptions of the various `MTYPE`s, including name, owning module, description, allocation method, and three allocation parameters. It also contains lists of memory types that are displayed for certain CLI features, such as the `show memory {protocol}` command.

Note: That the memory manager provides the implementation of the memory CLI. This prevents the library or other parts of the ZebOS ARS having to understand the underlying memory mechanism, and allows vendor specific extensions to memory management and monitoring. It additionally allows vendors to implement their own extensions to the memory CLI, or to remove parts of the memory CLI, if desired, without affecting other code.

Configuration Options

On supported operating systems, the configure script controls several features of the memory manager. On operating systems without support for the configure script, the `pal_memory.h` file contains the definitions for these features. There are several configuration options supported. When using the configure script, they can be controlled using `--enable-{feature}` or `--disable-{feature}`. When editing the file directly, either `#define` or `#undef` the appropriate switches in the first configuration section.

Feature name	Default	Feature description
memlog	disabled	The ability to log memory information, warnings, and errors, when it is possible (there are situations where it is not).
memmgr	disabled	The ability to allocate different `MTYPE`s using different techniques. This parallels the old `memmgr.c` file that was in the library before.
memchk	disabled	Checking of certain memory structures in an attempt to detect memory corruption, cell overruns, and attempts to manipulate invalid pointers (such as trying to realloc a stacked structure).
memacct	disabled	The accounting of bytes per cell and the ability to track the memory overhead imposed by the memory manager.
memacct-detail	disabled	Additional details to accounting, such as alloc and free request counting, and tracking lists of cells for all MTYPEs.

Two of these features are fixed in function: they directly control the behavior of the code. These two features are `memmgr` and `memlog`. The other three (`memchk`, `memacct`, `memacct-detail`) are mapped to certain features in the code.

memlog

The memlog feature allows the memory system to emit log entries for certain events. Based upon the other enabled features, there are many events that are logged when this feature is enabled. The memlog feature only increases the module size (and only one of the messages adds processor time overhead to the successful execution path). The memlog

feature is only able to log events that can be detected by the other configured features of memory manager (only a few events can be detected at all times).

Log Level	Log entry	Description
Debug	added supercell	An EXPRE type was out of space, so the memory manager added a new group of cells to its pool. This can occur if the memmgr feature is enabled.
Warning	invalid type	An invalid cell type was specified in a call requesting cell manipulation. This can always be detected.
	type mismatch	A call was made asking an existing cell to be manipulated; the call gave the wrong type for the cell to be changed. This can only be detected when cell type tracking is enabled.
	creating new cell from invalid one	A realloc request provided an invalid cell and the memory manager is creating a new one instead. This can only be detected when pointer checking is enabled or when header magic numbers are enabled.
Error	no memory	A call failed because there was not enough available memory to complete it. This can always be detected.
	none free	A call failed because there was no available cell for the `MTYPE` (prealloc method only). This can be detected only when memmgr is enabled.
	too big	A call failed because it requested a fixed size cell be larger than allowed. This can be detected only when memmgr is enabled.
	cell already free	A call requesting a cell be freed could not complete because the cell is already free. This can only be detected when header magic numbers are enabled.
	invalid context	A call requesting manipulation of a cell could not complete because that cell belongs to a different context.
	unknown method	A call requesting manipulation of a cell could not complete because the cell uses an unknown allocation method.
	cell address invalid	A call requesting manipulation of a cell could not complete because the address given is outside the valid address range for memory cells, or because the cell is not aligned correctly.
	cell header invalid	A call requesting manipulation of a cell could not complete because the address given did not point to a valid cell, or the cell's header had been corrupted.
	cell overrun detected	A call was made requesting that a cell be manipulated, but the cell data had overrun the cell size.
Critical	memory warning threshold reached	A call requesting creation of a new cell or expansion of an existing cell has pushed memory usage up to or above the warning level.
Alert	maximum memory size reached	A call requesting creation of a new cell or expansion of an existing cell has pushed memory usage up to the maximum limit, or would have pushed it beyond the maximum limit.

memmgr

The memmgr feature offers additional allocation methods. Currently, the system supports four methods: `HEAP` (standard heap alloc), `FSHEAP` (allocate fixed size heap cells), `PRE` (allocate only from preallocated workspace), and `EXPRE` (allocate from preallocated workspace, but expand that workspace if needed). Additional methods can be added to the code if desired. The `pal/api/pal_memory_desc.def` file contains the configuration mapping MTYPE to method. This adds a limited amount of overhead for all cell types, but any type that supports a free list (`PRE` and `EXPRE` do currently) can allocate new cells and free old cells very quickly after this small overhead, without having to make OS calls (providing, in the case of alloc with `EXPRE`, that there are cells on the free list). This feature adds a six machine word overhead per `MTYPE` to the static data.

Method	Description
HEAP	Standard heap based allocation of variable size cells. This method uses an extra machine word to track the size of this memory cell.
FSHEAP	Heap based allocation, but with fixed size cells. This has the advantage over HEAP of knowing the cell size without requiring additional data be stored with each cell. Because this method produces a fixed-sized cell, it allocs only enough space for statistics and for the memory requested.
PRE	Preallocation of a fixed number of fixed-size cells. These cells are maintained in two lists (in use and free) and can be allocated and freed in O(1) time. This method always allocate the maximum number of cells up front, and does not support addition of cells during runtime.
EXPRE	Preallocation of a base number of fixed-size cells. These cells are maintained in two lists (in use and free) and can be allocated and freed in O(1) time as long as there are cells available. This method can defer allocation of cells until they are needed, and can create additional cells (uses heap calls to build them) as needed after startup.

memchk

The memchk feature actually maps to various features that are controlled by settings in the source file. By default, memchk enables all of the additional fields per cell, as well as all of the accounting and request tracking features. It is intended to be used for debugging purposes (with `memlog`, preferably); disable it for normal builds because it imposes additional overhead in terms of runtime performance, runtime memory usage, code size, and static data size.

memacct

The memacct feature actually maps to various features that are controlled by settings in the source file. By default, memacct enables tracking of bytes in variable size cells, as well as allowing the memory manager to be aware of the overhead that it imposes upon the system.

memacct-detail

The memacct-detail feature actually maps to various features that are controlled by settings in the source file. By default, memacct-detail enables tracking of requests made (alloc, free, realloc) per `MTYPE`, and enables linking of all cells into lists (this feature allows remaining cells to be freed at shutdown).

Other Memory Manager Features

The `MEM_DEBUGGING_ENABLE` feature, not available from the configuration script, is intended for use only when debugging the memory manager itself, or diagnosing problems with modules that call it (or implement callouts). Use this feature with the memchk and memlog features enabled. Do not use it in a production environment, as it emits considerable output to the `stdout` device and adds substantial time overhead to the code.

Architecture

The ZebOS memory manager is designed to work in many environments. However, it must know about environment-specific variables. Because these are environment specific, rather than user options, none of these is addressed by the configuration script; they must all be edited in the `pal_memory.h` file itself. This allows platform specifics to be isolated from other parts of the code.

There are additional details about each of these options in the comments within the code.

MEM_GLOBAL_CONTEXT_ENABLE

Defining this indicates that it is safe on this platform to use a global context variable. Here, 'safe' means that the global variable is not shared between multiple instances of the PAL in different modules, or even in multiple instances of the same module. This saves one machine word per cell in header overhead, if `MEM_HEADER_ENABLE_CONTEXT` is defined. Disabling this requires changes to the interface functions at the bottom of the `pal_memory.c` file, so they can get the correct context.

MEM_GLOBAL_CONTEXT_[DE]REGISTER

In some non-protected operating systems, it is possible to ask the OS to protect a limited number of variables per process. This is often considerably less overhead than passing pointers on the stack, because the memory moves are only done once per thread switch rather than per function call. If such a feature exists, it also usually allows the global context feature to be enabled. These must be defined so they make appropriate OS calls to register a pointer as belonging to the local variables of the current context (`MEM_GLOBAL_CONTEXT_REGISTER`), or remove a variable from the same (`MEM_GLOBAL_CONTEXT_DEREGISTER`).

MEM_SEM_*

If using non-protected OS without the ability to protect even limited local variables per process, or if intentionally sharing memory space between processes for some reason, `MEM_SEM_TYPE` must be defined to be an OS semaphore or spin lock or other exclusion device handle type. If running protected OS, or the OS has the limited protection ability used above, and not intentionally sharing memory, `MEM_SEM_TYPE` must be left blank.

If `MEM_SEM_TYPE` is defined, `MEM_SEM_INIT` must be defined so it makes appropriate calls to create and claim (preferably atomically) the exclusion device. It must return a non-zero handle on success. If it returns zero or `NULL`, the memory manager interprets it as failure and abort its initialization process. If `MEM_SEM_TYPE` is undefined, this must be left blank.

If `MEM_SEM_TYPE` is defined, `MEM_SEM_DESTROY` must be defined so it makes appropriate function calls to release and destroy (preferably atomically) the exclusion device. If `MEM_SEM_TYPE` is undefined, this must be left blank.

If `MEM_SEM_TYPE` is defined, `MEM_SEM_REQUEST` must be defined so it makes a blocking, non-expiring call to claim the exclusion device. If `MEM_SEM_TYPE` is undefined, this must be blank.

If `MEM_SEM_TYPE` is defined, `MEM_SEM_RELEASE` must be defined so it makes a blocking, non-expiring call to release the exclusion device. If `MEM_SEM_TYPE` is undefined, this must be blank.

MEM_ALIGNMENT_REQUIREMENT

This setting has no effect if alignment checking is disabled and cell footers are disabled.

This is the alignment requirement for this platform, in increments of bytes. It must be set to an integral power of two, one or greater. This is usually the number of bytes in a machine word, but sometimes it is less (if a system with a narrower data bus than register file), or greater (if a system where the OS forces larger alignments to keep cache alignment). Set this to the machine word size, in bytes. If set to 1 (two to the zero power), it disables alignment forcing and checking.

typedef ... mem_int

This must be set so it defines `mem_int` to be an unsigned integer that is large enough to contain all possible pointer values. It is used as an override in pointer arithmetic to ensure there is no truncation or overflow created error.

typedef ... mach_word

This must be set so it defines mach_word as an unsigned integer with the same number of bits as an actual machine word. Under many cases, this is the same type of integer as `mem_int`. This is used in many structures to ensure proper alignment.

MEM_ADDR_*

`MEM_ADDR_LEN` is the number of digits to show when displaying addresses in hexadecimal, represented as a string (it is used in format strings). Normally, this would be the number of nibbles in the address, so eight on 32-bit systems (because it needs to be a number expressed as a string, “8”). On platforms using 64-bit addressing, it is possible that this would be sixteen (so “16”).

`MEM_ADDR_LINE` is a string comprised of hyphens, that is as long as the hexadecimal address display noted above. In the case of a 32-bit platform, “`--------`” is normal. On platforms using 64-bit addressing, it is possible that this would be longer (perhaps “`----------------`”).

MEM_*_MAXIMUM*

These settings control the default value for the memory usage limit, and whether the CLI can access the value. They have no effect if the cell header size field is disabled.

`MEM_DEFAULT_ALLOC_MAXIMUM` is the default initial value for the memory usage limit. This is based upon actual data usage (not the total OS memory exposure), and it is expressed in bytes. Set this to zero to disable this feature by default. It can be changed at runtime from the default value by the API, or from the CLI.

If `MEM_ALLOW_CLI_SET_MAXIMUM` is defined, the CLI views and sets the memory usage limit value (including disabling the limit). If not defined, only the API may access the memory limit, not CLI commands.

If `MEM_ALLOW_CLI_SET_MAXIMUM` is defined, then `MEM_ALLOW_CLI_SET_MAXIMUM_LOW` is the minimum value the CLI is permitted to use for the memory usage limit, and `MEM_ALLOW_CLI_SET_MAXIMUM_HIGH` is the maximum value the CLI is permitted to use for the memory limit. These settings have no effect upon the API.

MEM_*_WARNING*

These settings control the default value for the memory usage limit, and whether the CLI can access the value. They have no effect if the cell header size field is disabled.

`MEM_DEFAULT_ALLOC_WARNING` is the default initial value for the memory usage warning threshold. This is based upon actual data usage (not the total OS memory exposure), and it is expressed in bytes. Set this to zero to disable this feature by default. It can be changed at runtime from the default value by the API, or from the CLI.

If `MEM_ALLOW_CLI_SET_WARNING` is defined, the CLI views and sets the memory usage warning threshold value (including disabling it). If not defined, only the API may access the memory warning threshold and not the CLI commands.

If `MEM_ALLOW_CLI_SET_WARNING` is defined, then `MEM_ALLOW_CLI_SET_WARNING_LOW` is the minimum value the CLI is permitted to use for the memory usage warning threshold, and `MEM_ALLOW_CLI_SET_WARNING_HIGH` is the maximum value the CLI is permitted to use for the memory warning threshold. These settings have no effect upon the API.

Other Settings

Some compilers do not support the `inline` keyword. Several routines within the memory manager are defined as `inline` in order to improve performance. In an environment where code space is limited, or where the compiler does not support the `inline` keyword, set the definition of `INLINE` to blank. Under normal conditions leave `INLINE` defined as `inline`.

Some compilers do not support the volatile keyword. Under these conditions, set the definition of VOLATILE to blank. Under normal conditions, leave VOLATILE defined as volatile. Depending upon the optimization behavior of a compiler, this feature may be required (else corruption may result in the memory manager data).

Option Mapping

Some of the options that are accessed using the configure script (or through the first section of the pal_memory.h file) are able to have the features they control changed. These options are mapped to their actual features in the third section of the pal_memory.h file. The parts of this are controlled by the settings from the configure script or from the first section of the pal_memory.h file.

By default, the memchk setting (MEM_CHECK_ENABLE) turns on all of these features; the memacct setting (MEM_ACCOUNTING_ENABLE) turns on the MEM_HEADER_ENABLE_SIZE feature and the MEM_OVERHEAD_TRACK feature; the memacct-detail (MEM_ACCOUNTING_ENABLE_DETAILS) setting turns on MEM_HEADER_ENABLE_LINKS_HEAP and MEM_REQUEST_COUNT features.

MEM_HEADER_ENABLE_MAGIC

The memory manager supports checking the integrity of the memory cell by placing a ‘magic number’ in the cell header. This allows the memory manager to verify that cells are indeed valid, and to also know whether a cell is free or in use without having to traverse cell lists. Normally, this feature would be used when debugging. This adds only slight time overhead, but adds a one machine word overhead per cell in the memory manager.

MEM_HEADER_ENABLE_TYPE

The memory manager supports many memory cell types (MTYPEs). Under some conditions, different MTYPEs are allocated using different mechanisms and often use different size cell headers (some MTYPEs use headers and others not). By including the MTYPE for a cell in its header, the memory manager can validate that the cell being manipulated is the correct type, and avoid potential corruption that could be caused by manipulating a cell with an incorrect MTYPE. This adds only slight time overhead, but adds a one machine word overhead per cell in the memory manager.

MEM_HEADER_ENABLE_SIZE

The memory manager can account for memory usage per cell on fixed size cells without this feature, but in order to accurately track memory use on variable size cells, it must add a size field to the cell header. This feature also enables cell size details to be displayed for variable size cells in the CLI. This adds only slight time overhead, but adds a one machine word overhead per variable size cell in the memory manager (it does not affect overhead on fixed size cells). This also adds a two machine word per MTYPE overhead to the memory manager data.

MEM_HEADER_ENABLE_CONTEXT

In operating systems where memory space is shared between processes, it is possible to validate that memory cells belong to the particular process making the request. This feature can help to ensure that somehow memory cells are not being passed between processes (that could cause corruption within the memory manager) in such an environment. Under isolated environments, or if the environment is safe for the MEM_GLOBAL_CONTEXT_ENABLE feature, leave this disabled (in fact, it is disabled automatically if MEM_GLOBAL_CONTEXT_ENABLE is enabled). This adds only slight time overhead, but adds a one machine word overhead per cell in the memory manager.

MEM_HEADER_ENABLE_LINKS_HEAP

Some supported allocation methods keep cells in lists. This can be useful for debugging. This allows the memory manager to know all cells that are in the system of that type. Enabling this feature allows the memory manager to track heap cells that are not normally maintained in lists. There is a time overhead for this feature, as well as a two machine word per heap cell overhead. Allocation methods that normally keep cells in lists suffer the time and two machine word per cell overhead

whether this feature is enabled or not. The memory manager shutdown function can free any remaining cells that are in lists on final shutdown.

MEM_FOOTER_ENABLE_MAGIC

The memory manager supports checking memory cells for overrun conditions (where the data in the cell are larger than the cell itself). This feature adds a one machine word overhead (plus alignment waste) per cell, plus a slight time overhead. Normally, this would be enabled when debugging new code to make sure that it does not overrun the allocated cells for some reason (such as off-by-one bugs or similar), and disabled under other conditions.

MEM_CONTEXT_MAGIC_ENABLE

Similar to the checking of 'magic numbers' in the header and footer for a cell, the memory manager also supports checking for one in its own data space. This check is only made the first reference to the data during a particular call, but it is made per call. This adds a slight time overhead, and a one machine word overhead to the memory manager data.

MEM_POINTER_VALIDATE

The memory manager can track the valid memory range for cells, and also verify that any pointers provided to it are aligned correctly. This feature is intended to be used as a debugging aid; leave this disabled unless trying to track a problem that appears to be an invalid pointer. It adds a slight time overhead, and a two machine word overhead to the memory manager data.

MEM_OVERHEAD_TRACK

The memory manager can track its own overhead. This capability adds two machine words per `MTYPE` to the memory manager data, and slight time overhead.

MEM_REQUEST_COUNT

The memory manager can track the number of times each `MTYPE` is used for alloc, realloc, and free requests. This can be of assistance when trying to optimize the `MTYPE` settings. It adds a three machine word per `MTYPE` overhead to the memory manager data, and slight time overhead.

Memory TYPEs

The `MTYPE` specifications are expanded and moved. They now include the `MTYPE` name, the 'owning' protocol (the specifications list uses `IPI_PROTO_MAX` to indicate the library), a long description of the type as a clear text string, the allocation method, and three parameters for the allocation method.

The `pal_memory.c` file provides the `CELL_TYPE` macro that allows a standard interface to make these descriptions, even when certain features are disabled. The `pal_memory_desc.def` file in the `pal/api` directory contains the descriptions of the cell types and the needed includes for additional data such as the sizes of types to be used for prealloc class cells. It also includes lists of `MTYPE`s that are displayed by the `show memory` commands in the CLI.

pal_memory.def

The `pal_memory.def` file defines all of the standard PAL memory APIs, as well as defining all of the available `MTYPE`s. The enumerated type `pal_mem_t` includes all of the `MTYPE`s that are used in any of the modules. Please see the comments within the file for details about the function of each of the APIs provided. If `MTYPE`s are added, removed or otherwise changed in `pal_memory.def`, update the `pal_memory_desc.def` file.

pal_memory_desc.def (first section)

The first section of the `pal_memory_desc.def` file includes various header files needed when an `MTYPE` is defined to use fixed size cells. These header files are needed so the compiler knows the size of the cells at compile time (this can't be

changed after compile time). Please see the comments and includes within this section of the file for examples, to add additional headers to be able to make additional MTYPEs prealloc or expanding prealloc.

The pal_memory.def file also contains some additional memory related definitions used by the ZebOS code.

pal_memory_desc.def (second section)

The second section of the pal_memory_desc.def file defines a static constant structure that lists characteristics for MTYPEs. This structure is defined as an array, where excepting the first and last entries, order is not significant. The first entry must be MTYPE_UNKNOWN, and the final entry must be MTYPE_MAX. A limitation with this version of the memory manager is that MTYPE_ZGLOB cells are allocated in the heap (this should not be a problem, because this type is only allocated at startup). Allocate cell types that are variable in size in the heap, bacause prealloc class cells take up their maximum size. This structure is filled by using the CELL_TYPE macro. The CELL_TYPE macro takes these arguments:

```
CELL_TYPE (typename,owner,description,method,param1,param2,param3);
```

Argument	Description
typename	The MTYPE_* designation for the cell type.
owner	The IPI_PROTO_* designation for the owning module, or IPI_PROTO_MAX to indicate the library or the PAL.
description	A clear text string 30 characters or fewer describing the cell type.
method	The allocation method to use: HEAP, FSHEAP, PRE, EXPRE.
param1	The first parameter for the allocation method. See below.
param2	The second parameter for the allocation method. See below.
param3	The third parameter for the allocation method. See below.

Any MTYPEs that are not listed in this structure are assumed to be heap allocated with no additional parameters and no descriptions (they are displayed numerically when referenced by the CLI).

With the memory manager feature disabled, the memory system supports only one method, heap. All cells are allocated from the heap, and all frees are returned to the heap. When the memory manager feature is enabled, additional allocation methods are available. These methods may take advantage of the additional parameters provided, depending upon what they need.

Method	Parameter	Description
HEAP		Cells are allocated from the heap and immediately returned to the heap when they are freed. None of the parameters have any meaning, and leave each set to zero. Realloc functions as expected. Supports dynamic size cells.
FSHEAP		Fixed-sized cells are allocated from the heap and immediately returned when they are freed. These cells are fixed size. realloc change only accounting data because the cells can not change size. Any call that tries to alloc or realloc a cell of size greater than that specified here fails.
	param2	This is the cell size, in bytes.
PRE		Cells are allocated from a fixed size pool, and returned when freed. Only the first two parameters have any meaning, and leave the third set to zero. Once all cells are allocated, no more can be allocated until some are freed. Alloc and free take a fixed length of time. Realloc calls change only the accounting because the cells can not change size. Any call that tries to alloc or realloc a cell of size greater than that specified here fails. Only supports fixed size cells.
	param1	This is the number of cells to make available on startup. Only this many cells total can be allocated.
	param2	This is the cell size, in bytes.

EXPRE		Cells are allocated from a pool with an initial size, and returned to that pool when freed. All three parameters have meaning. After all cells are allocated, another allocate request attempts to expand the alloc pool by a set number of cells. Free takes a fixed amount of time to run. Alloc takes a fixed time to run unless there are no more cells, then it has to add more cells from the heap. Realloc calls change cell accounting because the cells can not change size. Any call that tries to alloc or realloc a cell of size greater than that specified here fails. Only supports fixed size cells.
	param1	The number of cells available at startup. The cells are allocated at startup whether any are used or not. If zero, the first alloc request initializes the pool with a number of cells equal to the additional cells parameter, and ignores this value.
	param2	This is the cell size, in bytes.
	param3	This is the number of additional cells that made available when an alloc request is made with no available cells. Use a minimum value of 16 for this parameter; for small MTYPES efficiency is gained by providing more cells; for large MTYPES, inefficiency could result from having much empty space.

Some reasons for using an alloc method other than HEAP would include:

- Performance: If an MTYPE tends to be allocated and freed a lot, the overhead for heap alloc and free can be huge. Using either PRE or EXPRE method can improve performance because these bypass the heap for normal operations.
- Memory fragmentation: If a particular MTYPE tends to be allocated and freed a lot, but not always freed in reverse order of allocation, or perhaps with intervening allocs and frees of other types, memory fragmentation can occur. Using either PRE or EXPRE method reduces this because these are allocated from and freed into reserved memory pools. This can also work for MTYPEs that are resized often, but will always be smaller than a fixed size.
- Tracking of memory usage: MTYPEs that use static sized cells do not require additional overhead in order to track their memory usage.

ZebOS ARS uses memory in several different ways. A few of these might make good candidates for using MTYPEs other than HEAP.

- Variable sized structures or strings. These are not good candidates for conversion to using PRE or EXPRE, because there would be considerable memory waste from cells that are not fully utilized.
- Variable sized structures that are always smaller than a fixed size. These might make good candidates for conversion to use PRE or EXPRE, but only if they are *never* allocated larger than the chosen size, *and* tend to be allocated, reallocated, or freed often. On PRE and EXPRE, because the actual cell size is fixed, realloc also runs in O(1) time and does not need to do memcpy even in the case where the cell is expanding.
- Fixed size structures. These are the optimal choice for conversion to PRE, FSHEAP or EXPRE allocation types, because they are fixed size. The best performance gain could be achieved by using the memory accounting details feature to collect statistics about the structures that are allocated and freed the most in a given configuration, and converting those.
- Buffers that are passed to the OS. If the OS frees these buffers, they are not good candidates, because the OS often cannot be asked to use the ZebOS memory manager. If, on the other hand, the OS returns ownership of these buffers to the application and the buffers have a reasonably small maximum size (perhaps some network MTU), they might be good candidates.

After using the memory accounting statistics to find the MTYPEs with high alloc, free, and realloc counts, decide on the best method for handling the MTYPEs. MTYPEs that always have a need for a few (never many) and that tend to see substantial alloc / realloc / free activity might be best using PRE. MTYPEs that have high alloc / realloc / free activity and tend to use more than a few cells (or may sometimes use none), may benefit from EXPRE. Use the FSHEAP method for byte-level accounting of a fixed size MTYPE without the additional machine word overhead for the byte count.

Several examples are given in the pal_memory_desc.def file of MTYPEs that use EXPRE allocation method, and many are given that use the FSHEAP method. Many of those using EXPRE used a similar method in the memory manager code in previous releases of the ZebOS ARS.

pal_memory_desc.def (third section)

The third section of the `pal_memory_desc.def` file defines several static constant structures that list the `MTYPE`s that are displayed by the memory manager when the CLI is asked to show memory for a particular protocol (or the library, or for all protocols). These lists include each `MTYPE` to be displayed for the protocols, and a list of these lists that is used by the `show memory` and `show memory all` commands.

Source Code

The memory module is written using standard C calls where needed, and calls to other PAL components or to the library where applicable. The code is in the `pal_memory.c` and `pal_memory.h` file, and the supporting code and configuration (`pal/api/pal_memory.def`, `pal/api/pal_memory_desc.def`) and some parts of make process files (`configure.in`, `acconfig.h`).

There were several design objectives for the memory manager:

Objective	Reason
Fast operation	Memory requests are made in large numbers by many products.
Small code size	It is important to keep the code small enough to run on embedded systems and in other applications that run on limited resources.
Small static data size	Same reason as small code size.
Accurate accounting	If accounting memory use, it is almost useless if there are inaccuracies built into the accounting system.
Various allocation methods	Sometimes there is a need to allocate and deallocate memory of a particular type repeatedly. In these cases, alternatives to (the often slow) heap helps system performance.
CLI compatibility	Several features in previous releases that had to be maintained, or at least, the UI had to be maintained.
Debugging aid	features that enhance debugging, such as trapping frees of already free cells, and warning of corruption, are useful.

The algorithms involved in implementing the objectives are portable, but some of the specific behaviors are not.

Additionally, extensibility was considered in the design of the memory manager. It is a reasonable process to add new `MTYPE`s and descriptions (that the CLI accommodates without changes to the CLI code -- even by the commands that accept an `MTYPE` name as a parameter), as well as to change settings for existing `MTYPE`s. Additional allocation methods are also possible, but it is a more complex process.

Keep the #define sections of the source at hand while viewing the code.

In laying out feature implementation within the code, the `#ifdef...#else...#endif` construct was used extensively. None of these constructs spans sections of the file, and they are usually only a few lines in length. Sometimes, they are layered, especially when behavior is different when features are used in combination (usually this happens in the CLI implementation).

CLI features

The older versions of ZebOS provided a simple memory accounting system. This system attempted to track active cell count and total byte count. Active cell count could be tracked with reasonable accuracy, but total byte count did not always work, particularly when dealing with cells types that could be different sizes.

There was also a ‘memory manager’ feature that handled various allocation methods. This feature could be enabled or disabled at configuration time.

This version replaces both of those features, and provides additional functionality. This version provides several CLI commands, that were provided by the old accounting function, as well as a few new CLI commands. This version also provides better accuracy, because each active cell size can be known, as well as the totals. Additional accounting details (such as the number of times an MTYPE has been allocated or freed, or the overhead for the memory manager itself) can also be enabled.

The original accounting CLI features are handled as they were before. Only the modules that were enabled during the build show up in the list, and only the options that are applicable to the running module are available (the RIP daemon cannot report data about the memory usage in the PIM daemon, for example).

There are several show commands (from view or enable state) provided by the memory manager, based upon the modules and features that are enabled.

Command	Description
show memory	Displays memory information for all modules
show memory all	Displays memory information for all modules
show memory bgp	Displays memory information for BGP module
show memory cspf	Displays memory information for CSPF module
show memory isis	Displays memory information for IS-IS module
show memory ipv6 ospf	Displays memory information for OSPFv3 (IPv6) module
show memory ipv6 rip	Displays memory information for RIPng (IPv6) module
show memory ldp	Displays memory information for LDP module
show memory nsm	Displays memory information for NSM
show memory ospf	Displays memory information for OSPF module
show memory pim	Displays memory information for PIM module
show memory rsvp	Displays memory information for RSVP module
show memory lib	Displays memory information for the library
show memory summary	Displays a summary of the memory manager status
show memory stats [MTYPE]	Displays statistics for all memory cell types that have active cells, unless the `MTYPE` is specified, then it displays statistics for only the specified `MTYPE`.
show memory stats all	Displays statistics for all memory cell types, used or not.
show memory detail [MTYPE]	Displays a list of active cells for all memory cell types that have active cells or have had active cells during this run, unless the `MTYPE` is specified, then it displays statistics for only the specified `MTYPE`. This command is only available if `MEM_ALLOC_TYPES_ENABLE` is defined (the `memmgr` feature is enabled) or if `MEM_HEADER_ENABLE_LINKS_HEAP` is defined.

Where an `MTYPE` is given, it can be either `MTYPE_*` style name, or the number assigned to the `MTYPE` in the enumeration of the `MTYPE`s. The list of known `MTYPE`s, and the valid numbers, can be displayed using the CLIs built-in help feature.

Note: The output from these commands varies according to the actual options used during the build. Appropriate notes are provided in Appendix 1 with the output samples.

There are also commands in the configure state of the CLI. These are intended to configure certain aspects of the memory manager. Note that these are available only if MEM_HEADER_ENABLE_SIZE and MEM_LIMIT_ENABLE are both defined. Also, these features must be enabled by defining `MEM_ALLOC_CLI_SET_MAXIMUM` or `MEM_ALLOC_CLI_SET_WARNING` as appropriate. If these commands are available, and the maximum or warning threshold is active, the `show memory summary` command displays the current maximum or warning threshold.

Command	Description
memory maximum <size>	Sets the memory usage maximum to <size>, specified in bytes. This command is only available if `MEM_HEADER_ENABLE_SIZE` is defined.
no memory maximum	Disables the memory usage maximum threshold feature.
memory warning <size>	Sets the memory usage warning threshold to <size>, specified in bytes. This command is only available if `MEM_HEADER_ENABLE_SIZE` is defined.
no memory warning	Disables the memory usage warning threshold feature.

API features

There are several additional APIs defined by the new memory manager. These are used to configure the memory manager, to check the memory manager status, and to provide notification to an external entity for certain types of memory events.

result_t pal_mem_warning_set(size_t warning);

This function sets the current value, in bytes, for the warning threshold. It returns `RESULT_OK` unless an error occurs, in that case it returns the error. Set the warning threshold to zero to disable this feature.

size_t pal_mem_warning_get(void);

This function gets the current value, in bytes, for the warning threshold. It returns zero if the warning threshold is disabled, or if an error occurs.

result_t pal_mem_maximum_set(size_t maximum);

This function sets the current value, in bytes, for the maximum memory limit. It returns `RESULT_OK` unless an error occurs, in that case it returns the error. Set the maximum to zero to disable this feature.

size_t pal_mem_maximum_get(void);

This function gets the current value, in bytes, for the maximum memory limit. It returns zero if the maximum is disabled, or if an error occurs.

size_t pal_mem_current_get(void);

This function gets the current memory usage, in bytes. It returns zero if an error occurs.

result_t pal_mem_handler_set(enum mem_event_type event, mem_event_handler handler);

This sets a handler in the memory manager for the specified event. It returns `RESULT_OK` for success, or the error on failure. Set a particular handler to `NULL` to disable the handler and to use the default behavior.

result_t pal_mem_handler_get(enum mem_event_type event, mem_event_handler handler);

This gets a handler from the memory manager for the specified event. It returns `NULL` if there is no handler or if there is an error.

Event callouts

The memory system supports several event callouts to notify other components of potential problems. Handlers may be registered for these callouts using the pal_mem_handler_set function, and may be retrieved using the pal_mem_handler_get function. Only one handler may be in place for a particular event, but different handlers maybe used for each event (or the same handler may be used if that is desired).

There are currently several events which may have handlers:

MEM_EVENT_WARN_THRESHOLD	**This indicates that a memory request which is currently pending would put total consumption at or above the warning level. This does not affect the results of the memory operation, unless the handler specifies the FAIL result.**
MEM_EVENT_MAXIMUM_LIMIT	**This indicates that a pending memory request would put total consumption at or above the maximum limit. Heap memory must be freed before the handler returns a result or the only result which will not FAIL is FORCE.**
MEM_EVENT_NO_MEM	**This indicates that a pending memory request is unable to be filled because the OS is not fulfilling a request by the memory manager for a heap cell. Heap memory must be freed before the handler returns a response other than FAIL or CONTINUE.**
MEM_EVENT_NO_FREE	**This indicates that a pending memory request is unable to be filled because there are no free cells of the type which has been requested. Memory of the particular type must be freed before the handler returns a response other than FAIL or CONTINUE.**
MEM_EVENT_TOO_BIG	**This indicates that a memory request has been made for a cell, but the requested size is too big for the requested type of cell. This is not recoverable and indicates either a configuration error or an error in the calling module. It will only FAIL (all results will FAIL).**
MEM_EVENT_INVALID_CELL	**This indicates that a memory request has been made to manipulate a cell which is not valid for some reason (such as invalid pointer, or corrupt cell header, or incorrect context). This is normally unrecoverable since it indicates either corrupt cells or an invalid pointer. It will FAIL for all except FORCE, which will make a new cell of the desired size.**
MEM_EVENT_INT_ERR	**This indicates an internal error in the memory manager, likely caused by incorrect configuration (such as a type using an undefined alloc method). This is not recoverable and likey indicates misconfiguration or some sort of corruption in the memory manager subsystem.**

The memory manager calls the registered handler (if there is none, it follows the default behaviour of assuming the MEM_RESP_CONTINUE result from the handler). The handler should, if possible, make an attempt to fix the situation.

Once the handler has performed its actions, it returns a response to the memory manager, letting it know what to do next. There are several responses. However, some problems are not recoverable and will always result in the memory manager failing the request.

Response	Event	Result
MEM_RESP_CONTINUE	**MEM_EVENT_WARN_THRESHOLD**	**Proceeed as if the warning threshold did not exist.**
	(all others)	**Return error.**
MEM_RESP_FAIL	**(all)**	**Return error.**
MEM_RESP_RETRY	**MEM_EVENT_WARN_THRESHOLD**	**Recheck usage against the threshold.**
	MEM_EVENT_MAXIMIM_LIMIT	**Recheck usage against the maximum limit.**
	MEM_EVENT_NO_MEM	**Retry request.**
	MEM_EVENT_NO_FREE	**Retry request.**
	(all others)	**Return error.**
MEM_RESP_FORCE	**MEM_EVENT_WARN_THRESHOLD**	**Proceed as if the warning threshold did not exist.**
	MEM_EVENT_MAXIMUM_LIMIT	**Proceed as if the maximum limit did not exist.**
	MEM_EVENT_INVALID_CELL	**Create a new cell of the specified size filled with zeroes.**
	(all others)	**Return error.**

In this context, 'Return error' indicates the normal mechanism of returning an error in `malloc`/`realloc`/`free` calls, as documented in the standard libraries.

MEM_RESP_RETRY can cause infinite loops: the memory manager retries the operation and if the situation has not been resolved, calls the same handler to attempt to get it handled. When using RETRY, be careful to avoid loops.

MEM_RESP_FORCE does not remove or disable the maximum limit, so forcing past it will require continued use of force for all allocations until the memory consumption falls back below the maximum limit.

All memory allocation or expansion requests that would put or keep memory usage above the warning level will cause the warning event.

Handlers for events that indicate a low memory or out of memory condition must not do anything which would allocate or reallocate memory, because they are reentrant functions; the new request might also be above the given threshold. All handlers may free memory, or make other requests that do not allocate or reallocate memory.

Memory event handlers use a `varargs` mechanism for addressing the state information from the memory manager. Currently, they all provide the same set of parameters, but this may change if new events are added, according to what data are available. The C prototype for a handler function is:

```
mem_event_resp_t mem_event_handler(mem_event_t, size_t, struct lib_globals *, pal_mem_t,
void *, size_t, size_t);
```

All events provide this set of parameters if they are available (if an argument is not available for the event, zero is used):

`mem_event_t` = the event type

`size_t` = the number of arguments which follow

`struct lib_globals` * = the library context for the memory manager (will be NULL on startup before the call to pal_mem_register, and may be NULL during the shutdown operations)

`pal_mem_t` = the cell type being manipulated

`void *` = pointer to the cell if it already exists (this is only valid on realloc and free)

`size_t` = number of bytes requested in the cell (this is only valid on alloc and realloc)

`size_t` = number of heap bytes free required to fulfill it (this is only valid on alloc and realloc) (this is only valid on realloc if the input cell was also valid)

Currently, the warning and limits are not aware of the memory subsystem overhead, but they are aware of the operation of the PRE and EXPRE types. Always free heap cells to free memory for NO_MEM, and always free the type being requested to free memory for NO_FREE.

CHAPTER 12 Timer

ZebOS provides a set of platform-abstracted timer functions in `pal_time.c`.

Function name	Description.
pal_time_start	This call starts the time manager.
pal_time_stop	This call stops the time manager.
pal_time_clock	This call gets the number of clock ticks since the system started; this call replaces `clock ()`.
pal_time_current	This call gets the current time. Returns the current time, plus sets the time_t at the end of the provided pointer (unless pointer is NULL then it only returns the current time). This replaces the `time ()` call.
pal_time_tzcurrent	This call gets time of day and timezone information. Puts current time, plus the current timezone in the provided space. Does not return time if `t` is NULL; does not return timezone if `tz` is NULL.
pal_time_mk	This call takes an expanded `struct tm` and compress it into a `time_t`. This call replaces the `mktime ()` call.
pal_time_gmt	This call takes a local time and convert it to GMT (UTC), in expanded form. This call replaces `gmtime ()`.
pal_time_loc	Take a local time and converts it into expanded form. This call replaces the `localtime ()` call.
pal_time_strf	This call takes an expanded time and converts it into string form. This follows the K&R definition for `strftime ()`.
pal_time_calendar	Convert the calendar time into string form. The calendar time is often obtained through a call to pal_time_current (); This function is used to replaced ctime ();

CHAPTER 13 Threads

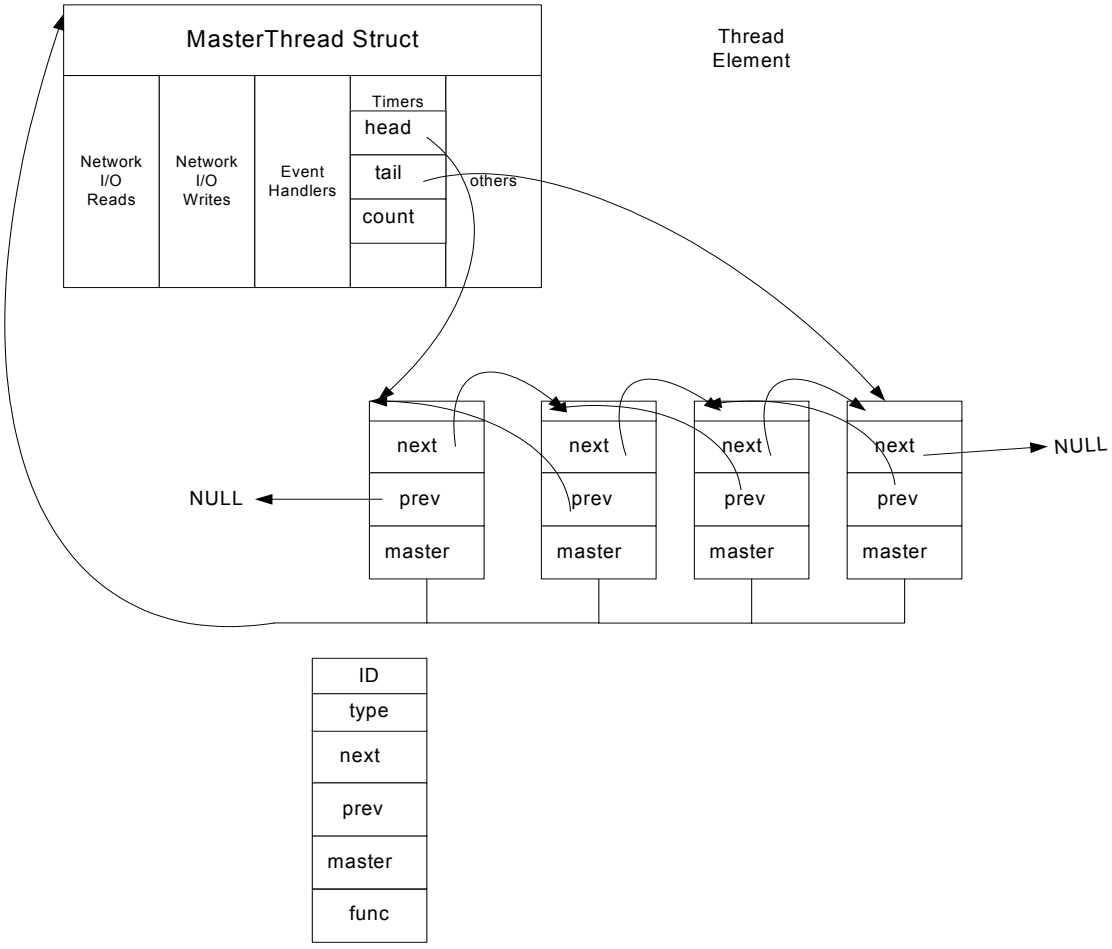
ZebOS ARS uses a thread-queuing mechanism in `/lib/thread.c` to manage a variety of asynchronous events. The threading mechanism the ZebOS daemon uses may be characterzed as a non-preemptive, execute-to-completion type of scheduling. As far as the host OS is concerned, it is a single thread. Therefore there are no contention problems, since all the code execution is serialized.

Each protocol module has a master thread structure to hold several thread linked-lists. Each linked list is dedicated for an executable type; network I/O read, network I/O write, handling an event, timer, etc. The thread linked-list is built of threading elements that contain pointers to the actual executable task. For every thread element, it has an executable function associated with it. For every task that needs to be executed, a thread element is added to the linked-list for its type. One important part of an event is the timer function. The main program waits on the main loop as:

```
while (thread_fetch (master, &thread))
    thread_call (&thread);
```

The function `thread_fetch()` uses the `select()` system call to perform a blocking wait on pre-set timers or any system I/O interrupt. When the `select` becomes unblocked, the callback is returned from `thread_fetch()` and the main loop executes the thread callback using `thread-call()`.

When several events arrive simultaneously (multiple timers, read, write), they are all placed into the EVENT queue; all such threads are continually provided from fetch until the EVENT queue is empty, in which case the blocked select will once again be invoked.



Internal Structure

ZebOS/lib/thread.h

```
/* Linked list of thread. */
struct thread_list
{
  struct thread *head;
  struct thread *tail;
  int count;
};
/* Master of the theads. */
struct thread_master
{
  struct thread_list read_pend;
  struct thread_list read_high;
  struct thread_list read;
  struct thread_list write;
  struct thread_list timer;
  struct thread_list event;
  struct thread_list event_low;
  struct thread_list ready;
  struct thread_list unuse;
  fd_set readfd;
  fd_set writefd;
  fd_set exceptfd;
  unsigned long alloc;
};
/* Thread itself. */
struct thread
{
  unsigned long id;
  unsigned char type;/* thread type */
  struct thread *next;/* next pointer of the thread */
  struct thread *prev;/* previous pointer of the thread */
  struct thread_master *master;/* pointer to the struct thread_master. */
  struct zglobals.*zg; /*pointer to the struct zglobals*/
  int (*func) (struct thread *); /* event function */
  void *arg;  /* event argument */
  union {
    int val;  /* second argument of the event. */
    int fd;   /* file descriptor in case of read/write. */
    struct timeval sands;/* rest of time sands value. */
  } u;
};
...
```

Functions in file ZebOS/lib/thread.c

Function Name	Description	Files (ZebOS/lib/)
Thread_make_master	Create a master structure that holds all thread linked-lists	Thread.c
Thread_add_read	Add a read I/O thread to the read linked-list	Thread.c
Thread_add_write	Add a write I/O thread to the write linked-list	Thread.c
Thread_add_timer	Add a timer thread to the timer linked-list	Thread.c
Thread_add_event	Add an event to the event linked-list	Thread.c
Thread_cancel	Cancel a thread to the linked-list	Thread.c
Thread_cancel_event	Cancel an event	Thread.c
Thread_fetch	Fetch a thread element to be executed from the master if there is any I/O and the timer expires.	Thread.c
Thread_call	Call a specific thread element	Thread.c
Thread_execute	Execute a specific thread element	Thread.c

Callback

When programmers want to handle an asynchronous event based on I/O or Timer Timeout, they pass a "Callback" pointer to the threading routines. When the event happens, control will be passed to the programmer-supplied "Callback" which can then handle the event. The callback pointer can be an I/O handler (using a file descriptor), or a time-out handler. The daemon is responsible for initiating these callback pointers. For example, from the zebos daemon:

- For Socket read: call `thread_add_read( zebosm, fd , callback );`
- For Socket write: call `thread_add_write ( zebosm, fd, callback );`
- For a Timer: call `thread_add_timer( zebosm, callback, <argument>, INTERVAL)`
- For an Event: call `thread_add_event (……)`

When the event occurs, the callback function will be invoked, and the <argument>, if any, is passed to the function. Within the CALLBACK, it is typical to re-trigger the event or to invoke `thread_cancel( )`. A typical macro within OSPF is `OSPF_TIMER_OFF (timer struct)`.

The `thread_fetch( )` returns any thread at the top of the event queue; then the top of the ready queue. Times are stored as actual (future) time.

The head of the list is subtracted from TIME-OF-DAY to get how long to wait. If there is no head of the list then the wait time is defined as infinity. During `thread_add_timer`(), the wait time is converted to actual time by adding Time-of-day to the time to wait. It is then sorted into the list of all other actual times. This wait time, and the functions `readfd`, `writefd`, `exceptfd` are used in the select call. If the read is satisfied, threads are taken from the read list and set into the event list. If the write is satisfied, threads are taken from the write list and set into the event list. All timer threads older than time-of-

day are taken from the timer list and set into the event list. If a thread event exists, its thread is returned. If there are no thread events then this process tries the select indefinitely.

CHAPTER 14 Runtime Configuration

These functions take care of reading and writing the runtime configuration. The run-time configurations are kept in `.conf` files (`bgpd.conf`, `ospfd.conf` and so on), the main one being `zebos.conf`.

pal_config_start();	none
pal_config_stop();	none
pal_config_open();	fopen for config files
pal_config_close();	fclose for config files
pal_config_read();	fread for config files
pal_config_reads();	freads for config files
pal_config_write();	fwrite for config files
pal_config_writes();	fwrites for config files
pal_config_seek();	fseek for config files
pal_config_pos();	pos for config files
pal_config_eof();	eof for config files

With PAL, the full-path, configuration filename can now be specified on the start up line:

```
nsm -f /usr/local/etc/myconfig.conf
```

Without a path

```
nsm -f nsm.conf
```

the NSM reads the configuration from the named file in the current directory. The command `write_memory` causes `file.old` to be written to the directory the configuration was read from.

How:

- code to read/parse config-filename as argument already exists. Filename is already passed to <prot>_main() functions.
- <prot>_loadconfig() functions call host_config_set() to store config_file in host->config[]. Then, they call pal_config_open().
- pal_config_open() calls host_config_get() to see if filename exists.
- If yes, it opens that file.
- If NULL, it opens the default file.

(Previously, it was just opening the default file always).

If pal_config_open() fails, the error is now LOGGED so that administrator knows what's going on. Previously, nothing was being done to handle this error. The `stderr/out` logging is turned off by default and this error is not displayed to the screen. It is logged to `/var/log/zebos/`.

`pal_config_open()` takes an argument that specifies the config file.

Run-time Search Order for Configuration files

1) If -f option is specified, configuration checks for this file ONLY.

2) Else, it looks for the default file in the following order:

1. (i) Check nsm.conf in CWD

2. (ii) Check nsm.conf.sav in CWD
3. (iii) Check nsm.conf in /usr/local/etc/
4. (iv) Check nsm.conf.sav in /usr/local/etc/

Note: This is example for nsm only. CWD = current working directory.

CHAPTER 15 Logging and Debugging

Logging

A log file can be created for each protocol by adding the debug command to the configuration file. For example, for OSPF,

```
log file <path/file>
debug ospf lsa
```

To tee the logs use the command terminal monitor for on, terminal no monitor for off.

```
plog_err
   zlog_debug
      _notice
      _info
      _warn
      _err
   keywords:
      packet
      lsa
      event
      nssa
```

For logging and debugging the user needs to be familiar with multiple commands such as ping, trace route, the many show commands, vtysh, netstat –rn, and the “logging” options in each of these commands. The routing modules also have the “-l” options for logging. The user will also need to know the use of vtysh option.

```
Telnet <host> <service>
```

where `<host>` is `localhost` or `<ip addr>`, and `<service>` is port Number or ZebOS alias defined in `/etc/services`.

Debugging

The zebos routing daemon supports all routing protocols. It is executed as:

```
zebos ( -bdhklrv ) ( -f config-file )( -P port-number )
```

where:

- `-b`, --batch: Runs in batch mode, parses its configuration and exits.
- `-d`, --daemon: Runs in daemon mode, forking and exiting from tty.
- `-f`, --config-file: Specifies the config file to use for startup. If not specified, it defaults to /usr/local/etc/zebos.conf.
- `-h`, --help: A brief help message.
- `-k`, --keep_kernel: On startup, don't delete self-inserted routes.
- `-l`, --log_mode: Turn verbose logging on.
- `-P`, --vty_port port-number: Specify the port that the NSM VTY will listen on. This defaults to 2602, as specified in /etc/services.
- `-r`, --retain: When the program terminates, retain routes added by zebos

`-v`, --version: Print the version and exit. If the zebos process is configured to output logs to a file, the ZebOS writes the `zebos.log` file to the directory that started ZebOS. The ZebOS process can log to a standard output, to a VTY, to a log file, or through syslog to the system logs.

Show Commands assisted debugging

Some of the simple “show” commands are:

show ip route; show ipv6 route; show interface; show ipforward; show ip6forward

Debugging can be accomplished by logging special categories, either to file or to the terminal or to both. For example, NSSA debug proceeds by first inserting into the configuration file `debug ospf nssa`. Then `enable` is issued followed by `terminal monitor`. Logs are viewed on the terminal console. The easiest category would be altering the CLI-Interface to include a rich variety of user-defined commands. (Debugging is easier under VxWorks because all routines are accessible, and harder under Linux because inter-task communication might be involved).

CHAPTER 16 Asynchronous Events

Unix

In the Unix environment, the ZebOS threads are idle until either a timer event triggers or until an I/O event ensues. These two operations are bundled into a single blocked select statement. Upon block-release the rules dictate that the program can use the FD_ISSET macro for each active File Descriptor to determine which I/O events are ready and can use the `gettimeofday()` on its sorted list of timers to determine which timers are ready. These conditions are actually queued up and their associated threads are executed, until there are no more active threads, in which case the program will fall into a blocked select once again. The above sequence is the same, in parallel, for all active, running daemons or processes.

RTOS

In the RTOS environment, the simulated select statement is NOT BLOCKED. An external process can call a ZebOS task to poll its I/O or poll its timer list to find any active thread ready to run. The process that calls ZebOS tasks can be in a multi-task environment, a single-task environment, or process-type environment (Unix). The method a process uses to call ZebOS tasks can be any of the following:

- Background Manager
- I/O Manager
- 1 Second Timer Tick Manager (i.e. RTOS_DEFAULT_WAIT_TIME)
- Timer Wake-up Manager

Current Main Loop

Each daemon performs its various forms of initialization and then enters a Main loop.

Example:

```
zebosm->master = thread_master_create();     Builds the Master thread struct
while (thread_fetch(master, &thread))
{
   thread_call (&thread);
}
```

`thread_fetch( )` blocks on a select statement until either the current time-out ensues or until any I/O operation unblocks. For the duration of the select, the task is idle. Upon select becoming unblocked, the callback is returned from `thread_fetch( )`, and the main loop executes the thread callback using `thread_call( )`. In case several events arise at the same time, (multiple timers, read, write) they are all placed into the EVENT queue; all such threads are continually provided from fetch until the EVENT queue is empty, in which case the blocked select will once again be invoked. ZebOS uses two priority queues; Events is for Internal Events, serviced first, and Ready is for I/O and timers. VxWorks emulates the select function by invoking `vs_select` in `vs_list.c`.

`CURRENT gettimeofday( )`

Unix

```
#include <sys/time.h>
#include <unistd.h>
gettimeofday( . . . );      /* standard UNIX function */
```

VxWorks

```
int
gettimeofday(struct timeval_L * tv ,void * unused)
{
struct timespec tp;
clock_gettime (CLOCK_REALTIME,&tp);     /* standard VxWorks function */
 tv->tv_sec=tp.tv_sec;
 tv->tv_usec=tp.tv_nsec*1000;
       return 0;
}
```

RTOS Methods

A ZebOS task can be the zebos daemon (the NSM running process and threads) or any protocol daemon (ripd, ospfd bgpd, and so on).

Under RTOS, each of the tasks above must be individually managed by the central I/O in the system and the central timer functions. One entry (or message) to the above tasks is called `<Task>_tic( );` ZebOS finds out what type of service is being called by looking at FD-bits or by looking at `gettimeofday( )`.

OSPF Threads

The following THREADS (t_* functions) are defined throughout the system:

t_router_id_update	t_router_lsa_update	t_distribute_update	t_spf_calc
t_ase_calc	t_abr_task	t_asbr_check	t_external_lsa
t_maxage	t_maxage_walker	t_lsa_refresher	t_router_lsa_self
t_poll			

These threads are usually associated with timers, to allow Queuing processes and Simultaneity processes to proceed.

CHAPTER 17 Porting ZebOS to Real-time Systems

Currently IPI has ZebOS software ported and running on all the major Unix platforms and embedded RTOS systems on the Intel/PowerPC/MIPS platforms. The customer can use any one of these software builds as a reference build. The simplest porting involves many steps, and many complicated situations could arise depending on the types of variations from the above reference builds.

The communication stack is designed to be system independent and easily transferred to most real-time platforms. The purpose of the real-time ZebOS is to develop efficient and light routing blocks with the following benefits:

- Portability
- Efficient utilization of a real-time environment provides high-performance routing.
- Scalability
- Integration with various physical media (for example, fiber optics, Ethernet, Serial Links, etc.).

When considering of porting of ZebOS to a RTOS, the following factors should be considered:

- Hardware16/32/64-bit CPUBig/Little Endian architectures Global Address Space (No Memory Management Unit)
- Software Real-time preemptive/non-preemptive
- OS Persistent File system or non-volatile memory (Flash memory, PCMCIA memory cards)
- TCP/IP stack
- Preemptive OS Scheduler
- Flash data storage
- SNMP management interface
- TTY management console ZebOS basic IP stack

The real-time ZebOS is implemented in the SendBox environment. ZebOS runs in a type of virtual environment, surrounded with the following blocks that interface with the real OS services and other interface:

- Physical Low-level Interfaces
- Inter Task Communication
- Configuration and Management
- TCP/IP stack

Most real-time operating systems provide the preemptive Scheduler with a guarantee of task switching. ZebOS runs efficiently in preemptive systems and provides these OS-specific, task-scheduled services:

- Guarantees the time interval for minimum task switching
- Watch Dog timer hooks
- Resistance to priority inversion
- Cold and warm system restart
- Hooks for handling power-off correctly

This task scheduling policy has fewer restrictions than preemptive schedulers. Behavior of non-preemptive ZebOS is similar to the Unix version of the routing stack.

Some of the following information has been discussed earlier. However it is useful for porting of ZebOS to other RTOS.

CHAPTER 18 Miscellaneous

Crypto

First of all, IPI recommends avoid using the plain `crypt()` function because it uses a static buffer for the return value and therefore isn't thread safe. The `des_fcrypt()` routine provides the same functionality, and because it takes the result buffer as a parameter, it is thread safe.

File and I/O management

PAL handles all file and I/O management to make sure that all protocols are not dependent upon any system-level details.

Globals

ZebOS uses the minimum number of global variables as possible, utilizing pointers for inter-process communication as parameters in function calls. The file `lib.h` contains the definitions for the global structures `typedef struct lib_globals`.

Index

N

O

P

R

S

T

V

Z