System control structure of the IBM eServer z900

by F. Baitinger
H. Elfering
G. Kreissig
D. Metz
J. Saalmueller
F. Scholz

As computer systems become more complex, the use of embedded controllers for initializing and maintaining system operation is becoming increasingly prevalent. In the IBM eServer z900, a new control approach was introduced. This paper discusses why its introduction was necessary and outlines its associated, key technological and economic innovations. In particular, the following topics are addressed: service subsystem topology, hardware elements for performing system control, hardware abstraction, object-oriented framework for control, and inter-networking of system control microprocessors.

1. Introduction

Traditionally, during the power-on phase of computer systems, CPUs start to execute instructions and initialize the systems into a state from which the operating system can be loaded. In addition to executing user applications, the operating system also runs applications that are needed to keep the system functioning. These applications, also referred to as system-control tasks, are responsible for monitoring system integrity and process any errors that might occur during operation. Usually, there is only one operating-system image controlling all aspects of system management. This type of system control is typically referred to as in-band control or in-band system management.

The exponential growth of computing requirements has resulted in the creation of larger, more complex, systems. Power-on and initialization of these large systems up to the point at which the operating system is fully available can no longer rely only on the system CPU. Instead, systems incorporate "helpers" (e.g., embedded controllers) that facilitate the initialization of the system at power-on. However, during power-on of these complex systems, critical errors can occur, which would prevent loading the host operating system. In the initial case in which no operating system is available, a mechanism is required for reporting errors and performing system management functions. Furthermore, given the diversity of user applications, it is no longer true that one operating-system image controls the entire system. At the high end, today's computer systems are required to run multiple different operating systems on the same hardware. A single instance of an operating system is no longer in full control of the underlying hardware. As a result, a system-control task running on an operating system which is not under exclusive control of the underlying hardware can no longer adequately perform its duties. For example, what would happen if a control task, in the course of recovering from an I/O error, were to decide to reset the disk subsystem? Data integrity might no longer be guaranteed for applications running on another operating system on the

0018-8646/02/$5.00 © 2002 IBM

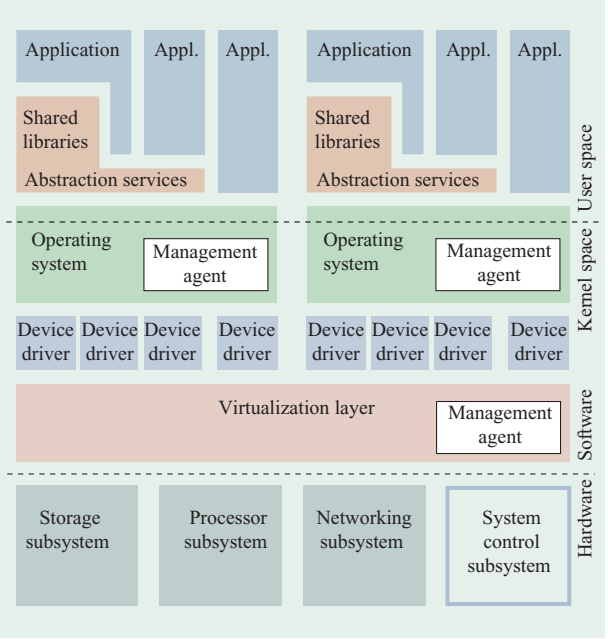

Figure 1

System software and hardware components.

same hardware. As a solution, system-control operations of a large system are moved away from the operating systems and are now integrated into the computing platform at places where full control over the system remains possible. System control is therefore increasingly delegated to a set of other "little helpers" in the system outside the scope of the operating systems. This method of host OS-independent system management is often referred to as out-of-band control, or out-of-band system management.

Figure 1 shows the software and hardware structure of such a system. The virtualization layer is the part of the system that has full control of its underlying hardware. This layer presents an abstract view of the underlying hardware to the operating systems. The embodiments of this abstract view are determined by out-of-band configuration tasks. All out-of-band operations are visualized in the box labeled System control subsystem.

Figure 2 shows a high-level view of an IBM eServer z900, together with its associated control structure. The system depicted to the left is composed of "cages." A cage can be a central electronic complex (CEC) cage or an I/O cage. The cage at the top is a CEC cage; it contains a set of CPUs forming an SMP system together with its cache structure, memory and cache control, and the memory subsystem. In addition, the CEC cage contains an I/O hub infrastructure. A system may contain one or more such cages. The cage in the center is an I/O cage, which facilitates I/O fan-out by linking the I/O cage to the CEC cage on one side and by providing bus bridges for the I/O adapters on the other side.

In addition to the functional structure, the figure also shows a system-control infrastructure that is orthogonal to the functional structure. The system-control structure is divided into management domains or management levels:

- *Management Level 1 domain (ML1):* Actuators and sensors used to perform node-control operations.
- *Management Level 2 domain (ML2):* Set of functions that is required to control a node:
  - Limited to strict intra-node scope.
  - Not aware of anything about the existence of a neighbor node.
  - Required to maintain steady-state operation of the node.
  - Does not maintain persistent state information.
- *Management Level 3 domain (ML3):* Set of functions that is required to manage a system (local to the system):
  - Controls a system.
  - Is the service focal point for the system being controlled.
  - Aggregates multiple nodes to form a system.
  - Exports manageability to management consoles.
  - Implements the firewall between corporate intranet and private service network.
  - Facilitates persistency for
    - Firmware code loads.
    - Configuration data.
    - Capturing of error data.
- *Management Level 4 domain (ML4):* Set of functions that can manage multiple systems; can be located apart from the system to be controlled.

Each CEC or I/O cage contains two embedded controllers called cage controllers (CCs), which interface with all of the logic in the corresponding cage. Two controllers are used for each cage to avoid any single point of failure. The controllers operate in master/slave configuration. At any given time, one controller performs the master role while the other controller operates in standby mode, ready to take over the master's responsibilities if the master fails. As a master, the CC performs the following functions out of the ML2 domain:

1. At power-on, determine configuration by reading the vital product data (VPD) from the inter-integrated circuit (I^2C)-attached serial electrically erasable programmable read-only memory (SEEPROM), including

- Cage type (CEC cage, I/O cage, etc.).
- Number and type of components present in the cage.
- Interconnect topology.

2. Initialize the functional hardware to a predetermined state by scanning start-up patterns into the chained-up latches using JTAG (Joint Test Association Group, IEEE 1149.1 boundary scan standard) or other shift interfaces.
3. Initiate and control self-tests of the logic circuitry.
4. At run-time, monitor and control operating environmental conditions such as voltage levels, temperature, and fan speed, and report any error conditions to system-management entities. In case of critical conditions, directly initiate preventive measures (e.g., emergency power-off) in order to prevent safety hazards.

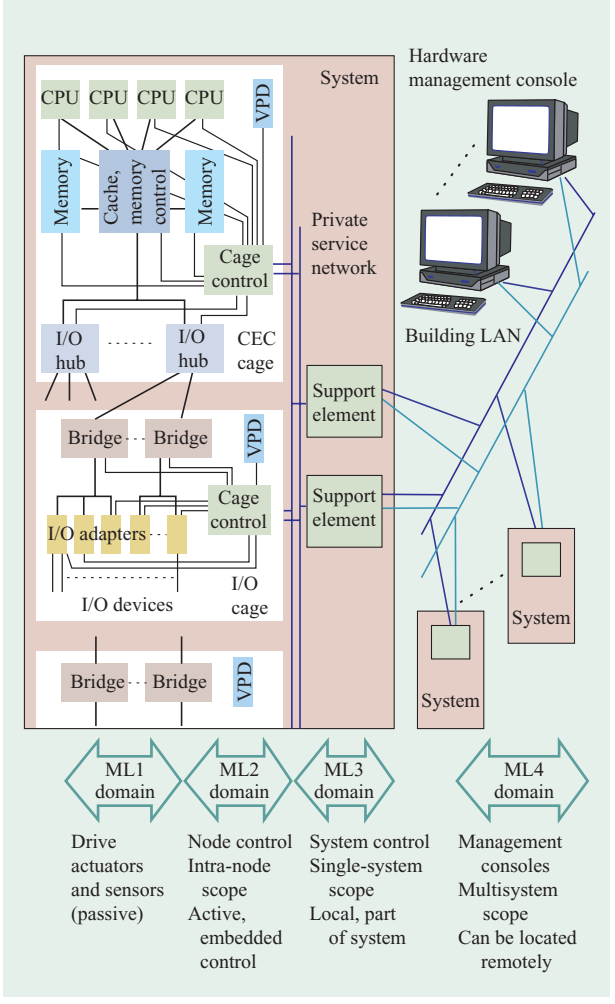

Figure 2

System control structure.

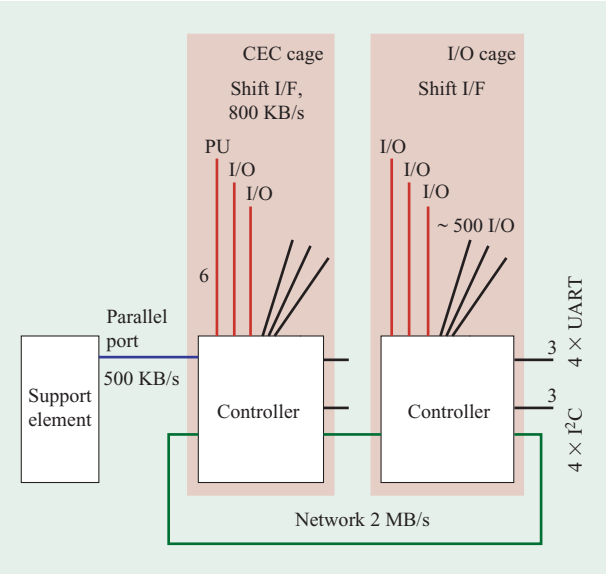

Figure 3

Control structure of previous systems.

In order to perform these functions, the embedded controller typically uses the following interfaces for intra-cage control:

- I^2C bus.
- GPIO (general-purpose I/O, sometimes referred to as digital I/O).
- UART (universal asynchronous receiver/transmitter, usually referred to as serial port).
- JTAG (Joint Test Association Group, IEEE 1149.1 boundary scan standard).

In addition to its intra-cage control scope, the cage controller interfaces with a higher-level system-control entity shown in **Figure 3** as the support element (SE). The SE operates in the ML3 domain of the system and is the point of system aggregation for the multiple cages. Traditionally the SE function is implemented on standard PC hardware (e.g., IBM ThinkPads* in recent systems). Communication between the SE and the CCs is facilitated via a private service network (PSN), which is based on standard Ethernet LAN technology running TCP/IP protocols. The entire structure (CCs, PSN, SE) is built with redundancy in order to avoid single points of failure. The SE is also the point of persistence for the system. The system configuration is stored on the SE disk. The SE executes the following tasks:

- Maintenance of configuration data [1] and system topology information.

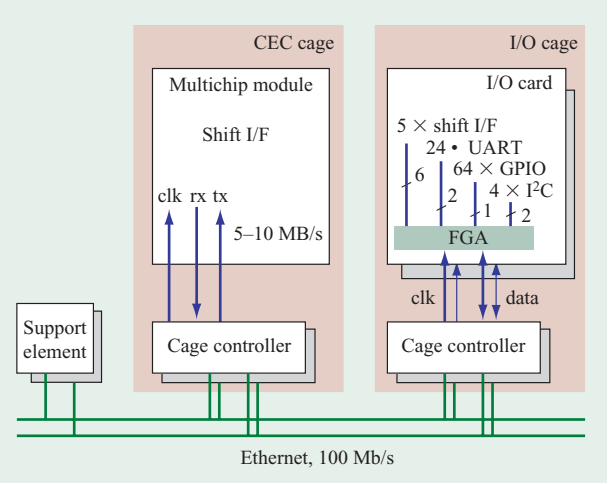

Figure 4

Control structure in z900 systems.

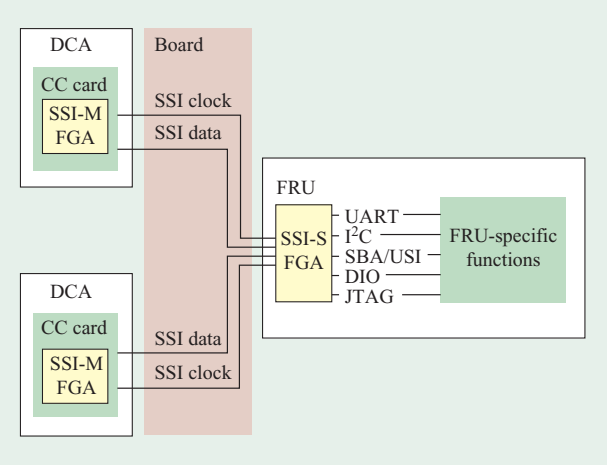

Figure 5

SSI interconnection between CC and FRU.

- Storing of code loads for host CPU firmware, I/O firmware, and cage controller firmware.
- Collecting and storing of error log information.
- Problem determination and reporting.
- Providing a graphical user interface (GUI) to conduct configuration and service tasks.

A z900 system is equipped with two redundant SEs. One of them operates as the primary SE, providing all system-control functions, while the other serves as the alternate SE [2].

2. Motivation and rationales

The overall concept of the control structure described here was also implemented in previous ES/9000* systems; however, a fundamental change was required for the z900 for the following reasons:

1. The controller technology used in the predecessor systems has reached the end-of-life state.
2. The significantly increased packaging density of the z900 could no longer be supported by driving the vast number of control interfaces directly from one point of control (see Figure 3 and **Figure 4**).
3. The connection from the CC to the SE was based on a PC parallel-printer interface which is length- and speed-limited.[1]
4. In the previous implementation, the single controller per cage was a single point of failure.

The required update also facilitated the replacement of proprietary implementations with a new structure based on open standards, such as POSIX** at the operating system interface, Ethernet for networking, and TCP/IP for communications. At the same time, single points of failure could be removed, and capability was provided to replace a cage controller while still maintaining the operational state of the entities it controls.

The primary reason for focusing on standards-based technologies, in contrast to what was done previously, was to be able to significantly reuse existing implementations of those standards.

3. Service subsystem topology

Cages in the IBM eServer z900 systems can currently contain as many as 38 field-replaceable units (FRUs). Each FRU is controlled by multiple interfaces, as shown in **Figure 5**. These interfaces are designed to support features such as upgrading of the configuration of a cage, or "hot-plugging" of FRUs in concurrent repairs.

For z900 system control, the CC is placed in the distributed converter assembly (DCA) of the power supply. This solution provides the following advantages:

- Each cage requires DCAs; if the CC is placed there, no extra slot space is required within the cage for the CC.
- The DCA is redundant.
- The CC is redundant.
- The DCA itself is a FRU. A failing CC can be repaired by replacing the DCA.

An identical CC is located in the bulk power interface (BPI) unit of the base power assembly (BPA), which is

[1] The SE is capable of a 2–3MB/s data rate via the parallel-printer interface, but the implementation of the parallel-printer port attachment in the previous CC was limited to 500 KB/s.

also implemented twice. The Ethernet hub is integrated in the BPA, since this position is central to the whole system. Two physically independent networks are provided in this way (see **Figure 6**). All CCs are connected to both networks, while each SE is connected to only one of the networks.

The CC microprocessor and several I/O components are realized as a system-on-a-chip approach. The processing unit is based on the IBM embedded Power401 core [3]. As shown in **Figure 7**, this processor core is equipped with two Ethernet media access controllers (EMACs) that are supported by a direct-memory access (DMA) engine. A programmable external bus interface unit (EBIU) allows the attachment of DRAM and flash-ROM-based memory as well as I/O devices to the controller. Timer functions and a universal interrupt controller (UIC) are facilities required by the operating system to support embedded control. A JTAG interface supports analyzing and debugging of the CC microprocessor remotely during code development and also later at the user's premises.

The large number of control interfaces in a cage requires innovative connectivity solutions. A new serial system support interface (SSI) is used for this purpose. The CC is connected to the respective FRUs by the SSI rather than by providing the multitude of interfaces mentioned above directly out of the CC microprocessor itself (see Figure 5). The number of interfaces and amount of wiring between the CC and each FRU are significantly reduced by using the SSI. To support the SSI on the CC and to provide the required types of interfaces on the FRU itself, a converter is required. This converter is realized in a second ASIC called an FRU gate array (FGA), which is configurable in two different modes of operation. In the role of an SSI slave function, it is located on each FRU and provides interfaces such as UARTs, I^2Cs, and DIOs on the respective FRU to serve the specific interfaces there. A set of registers is provided in the FGA for control of each interface. These registers are accessed by the CC like local I/O functions, but since they reside remotely on the FRU, the SSI link provides a remote-access method to these registers. An additional FGA device, configured as an SSI master, is attached to the CC memory bus. By the serial link it handles all register read and write operations to any SSI slave. For the purpose of problem isolation, there exists a point-to-point SSI link from the SSI master to the SSI slave on each FRU within the cage. This SSI link is also used for serviceability to detect whether a FRU is plugged in.

For redundancy reasons, two CCs are always implemented per cage. This implies that all FRUs must provide two SSI links—one to each of the CCs (see Figure 6). An arbitration mechanism within the SSI slave allows for concurrent access to the I/O interfaces from both CCs. A RAM array in the device provides a unique

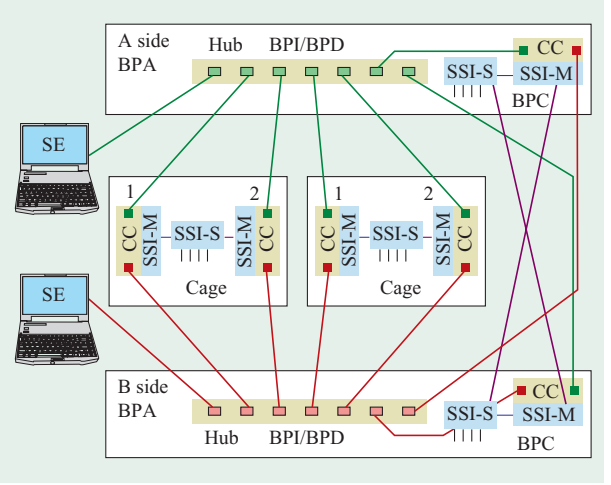

Figure 6

Network and intracage control topology.

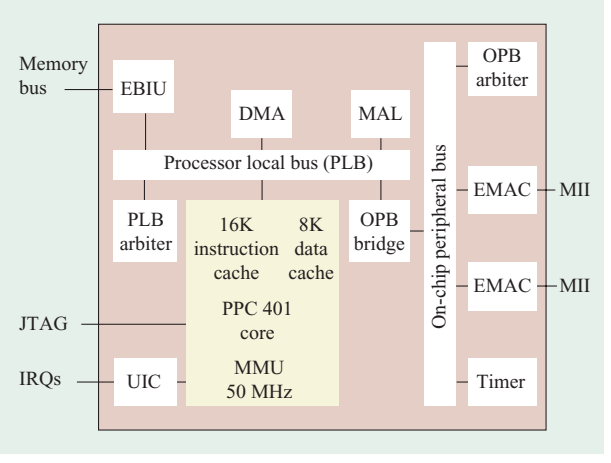

Figure 7

Cage controller processor ASIC.

storage area for vital state information in case the ownership for FRU processing is swapped between the CCs.

The same approach for FRU control by SSI is used for interconnection to the service interface of the CPU on the multichip module (MCM) via the clock chip located on the MCM. The dual SSI slave function, implemented on the clock chip, executes service operations on the CPU. Both SSI interfaces on the clock chip connect to the CCs of the CPU cage.

The full hardware redundancy of the service subsystem guarantees recovery from a failure of any single component

without affecting system operation and maintenance. Even multiple failures of different components (e.g., one network and a CC unit with all of its SSI interfaces) can be turned into a deferred-maintenance activity, avoiding unscheduled outages.

4. Hardware abstraction for intra-cage control applications

As outlined earlier, the intra-cage device access is the component in the z900 CC which controls the devices within a cage. For the applications, this component facilitates access to the devices. Most of these devices are located on remote FRUs and must be accessed through the SSI hardware (see Section 3).

The control applications have a functional interface to the device; i.e., they expect the devices to provide distinct services (such as reading a SEEPROM) without having knowledge about the implementation details. It is transparent to them whether a device is attached via SSI or natively, or whether a function such as "read scan-ring" is performed by a legacy I/O interface or directly through the shift engine of the CPU clock chip.

Software view of the intra-cage device hardware

The SSI architecture defines the control mechanisms between a microcontroller (e.g., based on the PowerPC Architecture*) and a multitude of different devices such as SEEPROMs, digital I/O lines, S/390* clocks, and other microprocessors. It comprises three major components: The *SSI master* is the interface to the processor and provides the communication paths to a set of slaves. The *SSI slave* contains multiple device engines; it is responsible for one FRU and drives all of its devices. The *SSI device engine* controls a device. From a software point of view, only the device engines are relevant, because they are used to control the device. They support a variety of devices, as listed in Section 1. As described there, the device engines are located on different cards and boards. The bridge between the processor and these devices is the serial communication link between the SSI master and its SSI slaves. One master can communicate with a large number of slaves. It even supports the hot-plugging of FRUs, which means that the card carrying an SSI slave can be inserted into or removed from a running system, and the SSI master will detect the event and inform the software by presenting an interrupt. The software can react to the interrupt by establishing an access path to the inserted device or by closing an open path.

Layers encapsulating device behavior

Each SSI engine has a corresponding engine handler in a controller software layer. The engine handler keeps track of the state of the engine and performs the state transition commands. A handler is also aware of the timing requirements of the associated engine and handles the asynchronous events triggered by interrupts from the engine. All handlers run in the context of the same device driver, which handles access to the SSI device and implements the serial communication protocol between the SSI master and the SSI slaves.

The interface to the engine handlers is facilitated by a device abstraction layer, which interfaces with the different devices. Engines with the same functionality can be accessed through the same function calls, even when their handlers are implemented differently. Because a scan-ring read can be performed by three different engines (CPU clock engine, I/O clock engine, or JTAG controller), three different engine handlers exist. All handlers use the same interface, called *readScanring(nrOfScanring, length)*, and only the address selection of the device determines which handler is used to fetch the data.

Application requirements

The device abstraction layer supports the accessing of a device without knowing the device internals. Each application uses a device interface that is independent of the different types of device attachment. It provides a functional representation of the attached device to the application, and a scan-ring access to the clock chip is done using a *readScanring(nrOfScanring, length)* function call. The interface call creates the necessary device control blocks and performs the device driver calls.

The application is not affected by changes to the hardware; devices can be attached differently, which requires different device driver calls. For example, an application calling *readScanring()* does not see whether the clock chip is attached directly to the SSI interface or whether an FGA chip is required to translate the SSI commands into legacy signals serving older clock chips. The association between the interface call and the device driver is done by a device "handle"; i.e., at open time, the device is identified by a name, and the open call returns a device handle. This device handle is used in all device operations and identifies the correct handler and device driver. This occurs at run-time, so that some configuration procedure or object can determine the current system setup and create the device handles, which are later used by the applications to perform the requested services without having any knowledge about the actual implementation of the device.

This scheme is flexible enough to include non-SSI-based devices, so that the whole device space can be covered by the abstraction layer. In addition, an atomic access path to the device is provided. There is no need to deal with the various concurrency problems that typically occur in multithreaded application structures.

Addressing scheme to select and access a remote device

The topology of the cards containing the FGA devices can vary from system to system depending on the configuration of the machine. In addition, this configuration can change at any time during execution, because most of the FRUs can be hot-plugged. This implies that the application working with the devices has to be aware of the current device configuration, must be able to create an active access path to the device, and must be aware of changes to the configuration.

The low-level communication protocol for communicating with a device is implemented in the SSI device driver. The appropriate register values to identify the FRU and the engine are derived from a control block, which is created when the access path to the device is established. In order to be able to create the access path, the application must perform the following steps:

1. *Select and open the SSI master:* Open the device driver and initialize the selected SSI master. This step returns the number of plugged ports, which indicates the available FRUs.
2. *Select and open an SSI slave:* Initialize the slave chip on the selected FRU and create a control block for that slave. This step returns a handle for the slave which identifies its control block.
3. *Select and open an SSI device engine:* Initialize the selected engine on a given slave and create a control block for it. This step returns a handle to identify the associated control block.

Subsequently, the application can use the handle to work with the device. The internal control block associated with this handle has all the information needed to find the device and to set up the low-level communication protocol, and it is passed to the SSI device driver. The control block also includes serialization mechanisms such as semaphores and mutexes to ensure atomic operation for one engine.

5. Object-oriented framework for FRU control

The applications on the CC provide information to the SE on the presence of processing units and other FRUs, their identification, and their state, such as powered on/off, activated/deactivated, and any of the different error states. In addition, they provide support to identify (via a light strip) the locations where the FRUs are plugged into the cage, and support to scan cage interconnections, for instance, via cable sensing. Furthermore, for each FRU the interfaces for initialization and service are provided to the SE. Other applications periodically ensure the correct functionality of the hardware, including a power-on self-test of the logic circuitry. For this purpose, the applications use the hardware abstraction layer to access the hardware, as described in Section 3. Errors detected during this hardware access are reported, and, if possible, a redundant path is chosen to maintain control over all installed hardware without any interruption. The main task of the control applications is to exploit the full benefit redundancy to achieve maximum availability of the hardware.

- *Redundancy in the ML1, ML2, and ML3 domain:* As described in Section 3, the z900 control structure not only has redundant CCs, but it also supports redundant paths to all FRUs. This means that even the path to the sensors and actors on the FRUs (I^2C, GPIO, UART, JTAG) is redundant. Figure 5 shows how this redundancy is achieved by connecting two FRU master chips to the same FGA slave chip. In other words, including the redundant SEs, the z900 is the first S/390 CMOS server with a fully redundant structure that spans the ML1, ML2, and ML3 domains, as defined in Section 1.
- *Object-oriented design on second-level controllers:* Another attribute of the new z900 control structure is that, by using a high-end Power-PC-embedded controller, state-of-the-art software engineering techniques such as object-oriented design and implementation could be introduced.

Object-oriented design

While examining the physical structure of the controlled hardware, the use of an object-oriented design for system control applications became apparent. The hardware exhibits physical aggregation, or containment, of parts in various places. These aggregations are expressed as so-called "has-a" relationships in terms of object models [4]. For example, a cage has several FRUs, a light strip, and cables. A FRU has chips, and a light strip has LEDs. The hardware domain also exhibits so-called "is-a" relationships that express different functional views of a physical part. For instance, a serial-channel card is a channel card which in turn is an I/O card. These "is-a" relationships are translated into the inheritance of attributes along a hierarchy of modeled objects.

Using object-oriented design encapsulates the attributes and functionality of the objects. It also simplifies the task of providing FRU state information. For example, in case of repair, all FRU information is available in the FRU itself, instead of being distributed in tables, lists, or global variables. In addition, it supports the introduction of new FRUs (cards). This is especially valid for new FRUs that share common properties with FRUs that already exist. Such FRUs can be supported more easily because they inherit a significant part of the requested functionality from FRUs that already exist.

Creation of the object model by FRU detection

The CCs of each cage both proceed into slave state after their initial booting. Then the topology service (see the section on boot and topology services) selects one CC per cage to become a master. Only the master controls the cage; hence, the configuration object model is instantiated on the master only. The main object, which contains all other objects, is called the cage object. The cage object is instantiated during the process of one of the two CCs taking over the master responsibility. For all possible plug positions, a place-holder object (the so-called unplugged FRU) is instantiated. Another object, representing the FGA master (SSI-M FGA in Figure 5), is instantiated. This object opens all SSI ports of the FGA master. When the SSI device driver is opened, it reports the presence of each FGA by evaluating the plug-detect feature of the SSI interface. The existence of a FRU is then known, but its identification is still missing. The related object for this state is called the unconfigured FRU. This FRU object does not have information about the exact configuration, in particular the wiring of the FGA or FRU, but it is aware of the default configuration, which allows accessing the SEEPROM on this FRU. When the unconfigured FRU has read the SEEPROM data, the identification of the FRU is known, and the unconfigured FRU is replaced by an object of the appropriate type. In some cases this object might be called a "mother FRU" object. Such an object represents a FRU that is a carrier for further, smaller FRUs, called "daughter FRUs." The daughter FRUs do not have their own FGA, but are controlled by the FGA of the mother FRU. If a mother FRU is detected, the code scans the GPIO plug-detect pins for the presence of daughter FRUs. If such FRUs are detected, their object is created immediately.

After all occupied SSI ports are scanned as described above, the entire SSI-controlled configuration is developed. This includes the DCAs or BPIs of the cage. Each cage has a minimum of two power FRUs. The power-control code must also have access to the FGA devices of the DCA or BPI. Therefore, the device handles of all of these FRUs are passed to the power control code. The power subsystem uses a UART protocol to communicate with all of its components. Having access to the UART devices, the power subsystem begins its initial scan. When this is done, the cage control code asks the power subsystem for its configuration. An object is then created for each of these FRUs, completing the initial buildup of the object model.

Experience with C++ as an implementation language shows that using C++ exception handling becomes a performance issue. This is the reason for using a lightweight method as the constructor of all objects. No resource allocation is performed in the constructors, since this may fail, and return-code handling is not possible in constructors. Therefore, each class has a build method which allocates the resources. This method does have return-code handling, so that error handling and recovery are possible during the initialization, even when C++ exceptions are not used.

Support for hot-plugging and exchange of FRUs

One of the challenges in the implementation of the class hierarchy was to support the hot-plugging of FRUs. This means that the design not only had to support the provision that a FRU could be plugged into the cage at any time, it also had to permit a FRU to be replaced by a new card, even of a different type. In a single-threaded environment, this would be done by simply adding two methods such as *add()* and *remove()* to the class of the cage object. In a multithreaded environment, care must be taken to make sure that an object is never deleted by one thread while other methods of the same object are still being executed by other threads. This is ensured by introducing a FRU manager. The FRU manager substitutes for the actual FRU object in the FRU list of the cage. The FRU-manager object is built for every possible plug position in the cage, and therefore the requirement for a lightweight object has to be met. Its only task is to control the lifetime of the FRU plugged into the plug position for which it is responsible and to provide access to this FRU object. Since the FRU manager is the only friend-class to the FRU classes, the methods of the FRU objects can be called only via the related access methods of the FRU manager. If a FRU object is currently being constructed or destroyed, any method call of the related FRU object is prevented; on the other hand, the destructor does not start until the method counter indicates that the object is no longer being used. For this, the FRU manager provides an exchange method. In the case of an unplug event, this method replaces a FRU object with an unplugged FRU. In the case of a plug event, the unplugged FRU is replaced with a functional FRU object according to its identification. During this exchange the old FRU object is destroyed, and the new FRU instance is created. While the exchange method is executed, the FRU manager prevents access to the methods of the FRU object by a simple mutex lock.

Automatic application restart

Applications in a high-reliability environment must minimize their downtime. Downtime can be the result of a hang condition, an out-of-resource condition, or a hardware failure. The current implementation is to perform a fail-stop whenever an unrecoverable failure is detected. Whenever a hardware error or an out-of-resource error is detected, the application on the controller is terminated by calling an abort function. This function gathers error information and copies it to a

special memory location before it stops the processor. Hang conditions are detected by a hardware watchdog timer or a communication time-out on the next level of control, and result in a stop condition. Whenever the CC runs into any of these stop conditions, the fatal-error recovery procedure is invoked; i.e., the CC reboots, and the code is restarted on the redundant CC and continues the work at the point of interruption. The same reboot-and-restart methodology is also used to apply a new code version to the CCs. In the predecessor generation to the z900, the controller was able to restart, but since it was not redundant, the code had to restart on the same controller. This means that controller hardware failure could not be recovered by employing a redundant controller.

A requirement known from previous systems is that the hardware has to keep its steady state; i.e., all of the output devices must retain their state, even if the code is no longer running. Furthermore, any interrupts must be latched during this time, such that they can be serviced after restarting the failing controller. A second requirement is that after a reboot, the CC must be able to retrieve the target state to continue the work at the point of interruption. This was realized by storing the target state of the cage in some memory space which is not reset by a reboot and is initialized only after returning from the standby power-off state. In this case, the target state and the real state scanned after a reboot can be compared, and the appropriate actions can be taken in case of a mismatch. In addition, due to the redundant design of the CC, this piece of memory must be accessible from both controllers. Therefore, using local RAM or flash ROM is not appropriate, since those components are dual-ported. But since any FRU must be accessible by both controllers, a special store register, the scratch-pad, in any FGA can be used for this purpose. A special bit called a "cold-start indicator" is used to ensure the validity of the data in the scratch-pad, and to indicate a power drop, which would invalidate the scratch-pad data.

This new structure not only allows the design to overcome hardware errors on the controller itself by exploiting the redundant controller via takeover, but also provides a second redundant interface to each FRU from the second controller. Thus, the complete path through the entire control structure into each FRU has no single point of failure, except for the controlled FRU itself.

6. Redundant private service network (PSN) with TCP/IP

The PSN inside the IBM z900 servers interconnects SEs and CCs, as shown in Figure 6. The SE acts as the central-service focal point for all hardware control applications [1] that communicate with the CCs inside the cages. Both SE and CC require a service network that provides peer-to-peer communication, as enabled by Ethernet [5] and TCP/IP [6] technologies. The following requirements are derived from the structure of this service network in order to support distributed control applications:

1. *Availability of the infrastructure in the presence of failures:* The dual-redundant and symmetric structure of the service network (Figure 6) is designed to facilitate the relocation of the function of failed parts to redundant parts with minimal interruption of service. All redundant parts are operated in "hot-standby" mode. In this mode, the redundant parts are started up to functional level, but only one of them is assigned to perform service tasks. Hot-standby mode offers the potential of fast fail-over because the state of redundant parts is known at fail-over time. This method enables instant recovery in the network domain, e.g., switching to alternate paths between controllers.
2. *Privacy of communication:* A service network must enable secure communication among the attached control processors. The CCs which directly control the system hardware interfaces must never be the target of malicious attacks. The system structure of the hardware management for z900 systems [2] establishes the SE platform as a firewall between the service network and any outside network. The TCP/IP configuration on the SE effectively disables forwarding of Internet protocol (IP) packets between LAN interfaces. This structure provides privacy without using cryptographic methods for authentication or encryption.
3. *Abstraction from the redundant capabilities:* Service applications executing across controllers require programming interfaces that hide the discovery and selection of network paths within the redundant structure.

As a result, the service network requires firmware in the following areas:

- Network configuration (see the next two subsections).
- Network management and monitoring (see the third subsection following).
- Network recovery (see the third subsection following).

TCP/IP configuration of the private service network

The redundant service network can be configured to either one of the following modes:

1. Two independent IP subnetworks.
2. A primary IP network and a backup network that is used only if the primary network fails.

An implementation of the hot-standby mode (see above) for all redundant parts implies that all networked

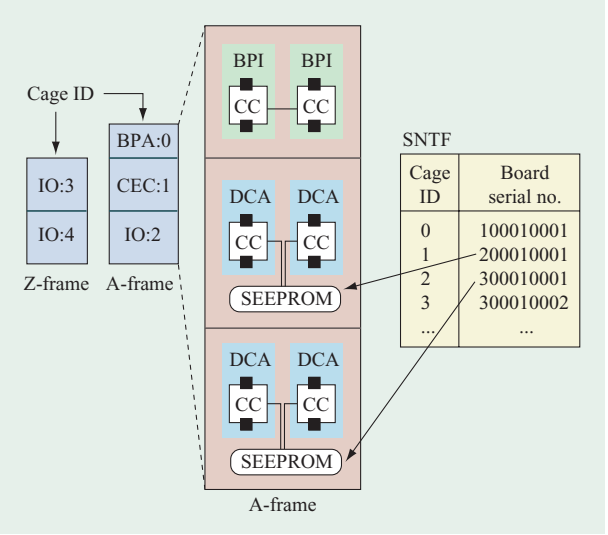

Figure 8

Retrieval of location information from controllers.

devices have to be periodically monitored. Hence, an IP configuration using two independent subnets is chosen. It enables the use of TCP/IP standard protocols such as ICMP [6] for network monitoring across the entire service network. If only one physical network were configured, monitoring of the unconfigured network could not be performed with protocols from the IP suite.

To minimize the risk of conflicts in the IP address spaces between the private service network and the customer network on the SEs, IP addresses from privately owned IBM address spaces are reserved. IP requires that all of the LAN interfaces of a computer must be configured using disjoint address ranges. The use of IP addresses from the nonroutable address ranges *10.0.0.0*, *172.16.0.0*, and *192.168.0.0* (see RFC 1918 [7]) would not have solved the problem, because the customer network could be purposely configured with the same range. Thus, the equivalent of two class C (up to 254 IP addresses [6]) reserved subnets was assigned to z900 development by IBM authorities.

An Ethernet network does not include the notion of orientation or spatial location. For instance, the mutual exchange of network cables from two stations at a repeater hub [5] does not influence the frame transport to these stations. However, the association of IP addresses with spatial locations of the configured device is considered a valuable feature for the z900 service network. The fact that specific CCs, for instance those CCs from the CEC cage, are always addressable with the same well-known IP addresses in all z900 systems enhances debugging of network traces.

In the z900 servers, unique identifications for the spatial locations of controllers are obtained from a data file. This service network topology file (SNTF) contains a table of the installed cages at manufacturing time. The stored data items are the board serial number of the cage and a unique cage ID that defines the spatial location of the cage according to the general floorplan of the z900 system. When CCs execute their startup code, they retrieve their board serial number from the board SEEPROM and put it into the boot request messages. The topology server (see the next subsection) on the SE matches the board serial number against the SNTF content to yield the cage ID assigned to the cage. For each cage ID, four IP addresses are pre-allocated for the network interfaces of the two CCs in a z900 cage.

Shown on the left in **Figure 8** is the floorplan of a z900 server with locations of cages and assigned cage IDs. The A-frame is expanded in the center of the figure, showing the access path from CCs to their common board SEEPROM. The bulk power interface (BPI) cage does not have a board SEEPROM. Instead, an auxiliary board serial number is constructed from the card serial number of the primary power interface.

In the z900 server, applications request connections to the master controller of a cage. By hiding IP addresses from applications, the selection of a path to a master controller can be delegated to an address-resolution mechanism. A new addressing mode is introduced that uses so-called "cage handles." A cage handle is an atomic and opaque entity for applications. It contains the cage ID and cage type in binary encoding. This scheme supports extensions when cages and packaging units for controllers are designed from the viewpoint of a one-to-many relationship. The firmware provides an API service (see the section on network resolution and communication API) to convert cage handles into IP addresses.

Boot and topology services

In this section, the requirements for configuration services are introduced. Boot and topology services are designed as SE-based network and configuration services. When CCs receive standby power, they execute the initial boot code from persistent local memory. This boot code requests an executable code load containing the operating system and applications from either SE. Both SEs execute a boot server that accepts the standard BOOTP messages (RFC 2131, RFC 1541 [7]) issued by the boot code. The vendor area of BOOTP messages is extended to accommodate special z900 data transport. The BOOTP message is also used by a failing controller to request the transfer of its memory dump to the primary SE.

The companion of the boot server is the topology server, which constructs a view of the cage configuration based on data obtained from BOOTP messages sent by

controllers. When a CC has successfully completed its initial startup, the topology server assigns IP addresses to the network interfaces of controllers according to the cage ID obtained for the cage from the SNTF.

The topology server exports a cage-oriented view to the functional SE code. A cage is reported as a functional unit to the SE when the topology server has been notified of the successful completion of the master role assignment of a controller in this cage. **Figure 9** shows the basic cooperation between boot and topology server.

The boot and topology servers execute on both the primary and the alternate SE [2]. This establishes a higher degree of fault tolerance and provides load balancing for the cold start of the entire system. For some special system scenarios (for instance, when applying firmware updates), the boot function on the alternate SE can be disabled. There are important tasks that can only be performed by the primary SE—for instance, the initial assignment of the master role[2] for one of the controllers in each cage. Figure 9 shows the election of a master controller by sending a "become master" message to one of the controllers in the cage with cage ID = 1.

When a SE is accidentally rebooted, the topology server loses its configuration because it is not persistently stored. However, the topology server on the restarted SE reconstructs its configuration by accepting InfoBoot [8] messages. These are periodically sent by each CC into all networks as "keep-alive status" messages. They have the same format and content as a standard BOOTP message but contain additional status information, e.g., whether the controller is executing in master or slave role. These InfoBoot messages also solve the problem of maintaining consistency between the data sets of both topology servers.

Network management and recovery services

Network management is the task of determining the status of a network configuration and applying explicit changes to that configuration. The characteristics of a z900 system demand that the status of a failed hardware part must be reported as early as possible to trigger a repair action. This requires periodic retrieval of status data from the entire service network. The definitive standard in the field of network management is the Simple Network Management Protocol (SNMP) [9] for large TCP/IP networks. For the z900 private service network, however, an even simpler and smaller set of management functions for automated control is considered more appropriate. Thus, SNMP functions are not used inside the z900 service network.

Instead, network diagnostic servers (NDS) and network management servers (NMS) are the firmware components

[2] The occurrence of two master controllers in the same cage must be prevented to avoid conflicts with respect to hardware access.

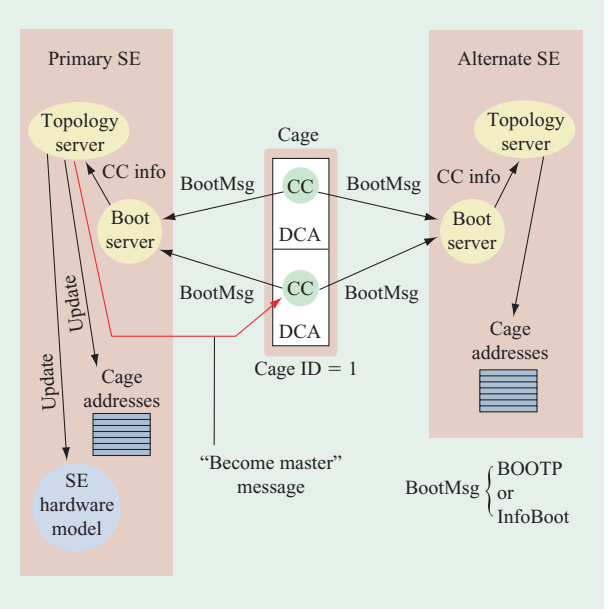

Figure 9

Topology servers receiving and updating CC information.

for network monitoring and management. They are executed on all CCs and on both the primary and the alternate SE. For each diagnostic request, a coordinating NDS instance is selected that executes on a controller with two configured and enabled network interfaces. All other NDS instances report the status of their local links to this coordinating NDS instance upon request.

The NDS uses classical TCP/IP methods on different protocol levels: It performs local interface checks by calling special *ioctl()* functions from the Ethernet device driver to retrieve the counters of broken Ethernet frames or excessive collisions. To determine the response behavior of remote controllers, ICMP ECHO [6] request packets are sent to these controllers. This ensures that the network paths to these remote controllers are working properly.

The NMS services provide functions for configuring IP routing tables, distribution of cage information, and activation or deactivation of network parts. All CCs are configured with IP forwarding [6] enabled. The NMS services preferably select a slave controller as the intra-service network router to relay IP packets across the service network. The asymmetrically attached SEs require IP forwarding to communicate across the service network. With the help of IP forwarding, a master controller with a deactivated or broken network interface retains the ability to communicate with the primary SE.

The network heartbeat worker (NHW) is the instance for coordinating recovery of network paths and master

controllers. It executes as a single instance on one of the cage controllers, preferably on an I/O slave controller. The NHW cannot be executed on a SE because its services are required when SEs are not available. The NHW can be restarted on any other controller if the previous instance has crashed. The NHW periodically invokes the NDS to check the network. If it detects a failed network link, it invokes the NMS recovery functions. If it detects a failed master controller in a cage, it triggers the CC reassignment function[3] for that cage. As a result, the companion slave controller is initialized as the new master controller. The NHW features an alert interface for applications (see the next subsection). Applications wait synchronously at this interface until a recovery has completed.

Network resolution and communication API

In order to satisfy the requirement to shield applications from the details of the redundant network, the network API is designed and implemented in the C++ [10] language. Basically, it consists of a TCP socket wrapper with the typical set of call interfaces [*connect()*, *accept()*, *send()*, *receive()*, *close()*] as defined in [11], but enhanced with additional functionality.

Address-resolution services map cage handles to the IP addresses of master or slave controllers. When invoked with the cage handle of a remote controller, the *connect()* call establishes a connection to one of the potential IP addresses of the master controller in the target cage if not otherwise indicated. Error codes from socket calls in [11] are mapped onto classes of simpler return codes—for instance, no network path to destination. If checking of a remote endpoint indicates a problem, the NHW alert interface is invoked. Upon return, the application is informed that the remote endpoint has changed and that it must be reset.

The network API is provided for execution on multiple platforms, OS-Open and OS/2* in the z900 implementation. It hides platform-specific details of socket calls from the application. For instance, it applies buffer sizes to *send()* calls according to the capabilities of the local operating system. When the caller passes a long timeout value by a *recv()* call, the network API applies an internal timeout value to periodically check whether the remote site is still responding.

Concluding remarks

The architecture of the system control structure of the IBM eServer z900 is the foundation for extensions to future zSeries systems. Its redundant control structure provides a foundation for the vision of self-management and self-repair. Efforts are currently underway in the IBM Server Group to seamlessly extend this structure into other products, e.g., iSeries and pSeries systems. During the implementation of the z900 system control structure, the advantages of developing the foundation on the basis of standards-based implementations became evident. In particular, the reuse of an existing POSIX-compliant control program and an existing implementation of a full TCP/IP protocol made it unnecessary to develop this functionality, thus permitting efforts to be more effectively directed at control and management aspects.

[3] The CC reassignment function can be triggered only by an instance that is not executing on any controller in the same cage. This design principle prevents an erroneous "self-appointment" of a slave controller that could lead to "ping-pong" situations.

Acknowledgments

The authors wish to thank Andreas Kuehlmann, Rudolf Land, and Steve Nichols for their helpful suggestions regarding the manuscript.

References

1. A. Bieswanger, F. Hardt, A. Kreissig, H. Osterndorf, G. Stark, and H. Weber, "Hardware Configuration Framework for the IBM eServer z900," *IBM J. Res. & Dev.* **46,** No. 4/5, 537–550 (2002, this issue).
2. B. D. Valentine, H. Weber, and J. D. Eggleston, "The Alternate Support Element, a High-Availability Service Console for the IBM eServer z900," *IBM J. Res. & Dev.* **46,** No. 4/5, 559–566 (2002, this issue).
3. For IBM Microprocessor products, see *http://www-3.ibm.com/products/powerpc*.
4. Grady Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin Cummings Publishing Co., Inc., Redwood City, CA, 1994.
5. Charles E. Spurgeon, *Ethernet—The Definitive Guide*, O'Reilly Publishers, Santa Clara, CA, 2000.
6. W. Richard Stevens, *TCP/IP Illustrated, Volume 1—The Protocols*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1994.
7. For IETF Request for Comments (RFCs), see *www.ietf.org*.
8. Friedemann Baitinger, Karlo Petri, Kurt Naegele, and Frank Scholz, "Extensions of the BOOTP Protocol Toward Automatic Reconfiguration," IBM Boeblingen patent application DE8-2000-0133, 2001.
9. William Stallings, *SNMP, SNMPv2, SNMPv3, and RMON1+2—Third Edition*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1999.
10. Bjarne Stroustrup, *The C++ Programming Language—Second Edition*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1991.
11. W. Richard Stevens, *UNIX Network Programming, Volume 1—Networking APIs: Sockets and XTI*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1998.

Received October 5, 2001; accepted for publication April 3, 2002

Friedemann Baitinger *IBM Server Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (baiti@de.ibm.com).* Mr. Baitinger is a Senior Technical Staff Member, currently working on IBM eServer support processor architecture and design. He received an M.S. degree in electrical engineering from the Fachhochschule Furtwangen in 1984. He subsequently joined IBM at the Boeblingen Development Laboratory, working on the development of communications subsystems for the IBM 9370 and IBM 9371 systems until 1991. From 1991 to 1994, he was on international assignment to the IBM facility in Poughkeepsie, New York, where he worked on the design and implementation of communication channels for the IBM Sysplex technology, which was introduced with the IBM 9672 system. Since 1995 Mr. Baitinger has worked on hardware systems management architecture and design, focusing specifically on the introduction of open-standards-based technologies and the elimination of single points of failure by providing redundancy for communication networks and microcontrollers.

Herwig Elfering *IBM Server Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (hge@de.ibm.com).* Mr. Elfering is a Staff Engineer, currently working as a team leader on the zSeries service processor application code for cage configuration and control. He received an M.S. degree (Dipl. Ing.) in electrical engineering from the faculty for Digital Information Processing of the University of Paderborn, Germany, in 1994. He joined IBM at the Boeblingen Development Laboratory that same year in the Power Control Department, where he worked in different areas of S/390 power and system control applications for the G2 to G6 systems. Since 1997, Mr. Elfering has worked on the design and implementation of the zSeries power and system control network, focusing on the reliability, availability, and serviceability of the IBM z900 eServer.

Gerald Kreissig *IBM Server Group, 11400 Burnet Road, Austin, Texas 78758 (kreissig@us.ibm.com).* Mr. Kreissig is an Advisory Engineer, currently working on IBM eServer service processor design. He graduated in 1979 from the University of Bonn, Germany, with an M.S. degree in computer science. Before joining IBM in 1981 at the Boeblingen Development Laboratory, he had a fellowship for postgraduate studies at the Computer Science Department of Purdue University. He began his work with IBM on VSE operating system development. From 1983 to 1987 he worked on operating system concepts in the Advanced Technology Group, which included a one-year international assignment in Dallas, Texas, to port UNIX System V to the S/370 architecture. Between 1988 and 1994, Mr. Kreissig contributed to most of the S/390 development projects in the IBM Boeblingen Laboratory, including those on the 9370, 9371, 9672 (CMOS) G1 and G2 systems. From 1994 to 1997, he developed an object-oriented framework for real-time applications in manufacturing systems. From 1998 to 2000, he worked on the design and implementation of the new system support architecture for the zSeries z900 server. Since 2000, he has been on international assignment at the IBM Server Group facility in Austin, Texas.

Daniel Metz *IBM Server Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (metz@de.ibm.com).* Mr. Metz is an Advisory Engineer, currently working as a Project Manager for zSeries microcode. He studied electrical engineering at the FHT Mannheim and graduated in 1994 with an M.S. degree (Dipl. Ing. FH). That same year, he joined the IBM Development Laboratory in Boeblingen to work on the design and implementation of a fail-safe S/390 system control structure based on transputer microcontrollers. From 1998 to 2000, he was a member of the System Control and Architecture Board, which developed the system support architecture for the zSeries z900 server. His responsibility and emphasis was on cage control and FRU configuration.

Juergen Saalmueller *IBM Server Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (saalm@de.ibm.com).* Dr. Saalmueller is an Advisory Engineer, working on the eServer service subsystem. He graduated in 1982 with an M.S. degree in physics from the University of Ulm, Germany. He received a Ph.D. degree in physics in 1987 from the RWTH Aachen, Germany, where he received the Borchers Award for his thesis on optical studies of semiconductors. Dr. Saalmueller joined IBM at the Boeblingen Development Laboratory in 1987, working on integrated I/O adapters for the 9370 series. Starting in 1992, he developed a series of adapter cards for digital wide-area networks, delivering the world's first PC card for such applications in 1994. From 1996 to 1997, working with a small team, he developed an ATM multiplexor unit for switched digital networks for the NWAYS development organization in La Gaude, France. Dr. Saalmueller joined the S/390 service subsystem hardware development team in 1997, working on the implementation of a new service infrastructure for the zSeries z900 server. He is currently working on the definition and design of a new service infrastructure for the next generation of eServers.

Frank Scholz *IBM Server Group, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (fscholz@de.ibm.com).* Dr. Scholz is an Advisory Engineer, currently working on architectures for integrated service networks, with an emphasis on high availability. He received M.S. and Ph.D. degrees in computer science from the University of Karlsruhe, Germany, in 1976 and 1981, respectively. Dr. Scholz subsequently joined IBM at the Boeblingen facility, where he has contributed to several IBM products in the UNIX domain (IX/370, AIX/ESA), and to efforts on cluster computing and parallel programming (CERN Parallel Programming Compute Server). Since 1996, he has been working on the system control of high-availability service networks. Dr. Scholz was the leader of the team that developed the design of the service network of the IBM eServer z900.