z990 NetMessage-protocol-based processor to support element communication interface

C. Axnix
E. Engler
S. Hegewald
T. Hesmer
M. Kuenzel
F. M. Welter

The communication interface between support element applications and applications running on the zSeries® system processors is an essential part of the zSeries system design. For example, the interface is used to load firmware during startup, it is used for service actions such as configuring or deconfiguring I/O channels, and for many other functions. It must be fast, reliable, and failsafe. A special hardware interface in the clock chip is used to connect the service infrastructure (support element and cage controller) to the central electronic complex (CEC). Four firmware parties are involved in the communication: support element, cage controller, and two firmware layers running on the processors in the CEC: millicode and i390 code. Starting with the z900, the interface between the support element and cage controller was implemented using the NetMessage protocol, whereas the interface between the cage controller and processors still used the legacy service-word communication protocol from previous IBM S/390® models. This meant that the cage controller had to translate the NetMessage protocol from the support element side to the legacy service-word protocol toward the CEC side. In the z990, the communication interface between the support element and the CEC was generally replaced by the NetMessage protocol. The following paper describes the new design and structure of the support element to CEC communication.

Introduction

The IBM z990 server is a large-scale server designed to meet the needs of customers at the high end of the marketplace. Such servers are capable of running multiple operating systems at the same time. The maintenance and management of these servers is done concurrently. With the IBM z900 server, an out-of-band system control structure was introduced to manage these complex systems [1]. Such management tasks include testing the hardware before the operating system is loaded, loading the operating systems, concurrent repair, concurrent upgrade, reporting of and recovering from errors, etc. To accomplish these tasks, a distributed service and control subsystem consisting of redundant support elements, cage controllers, and communication links has been introduced (**Figure 1**). (A more detailed, in-depth description of the system control structure can be found in [1].)

The firmware responsible for executing system management tasks runs on different system components (the processor module itself, the cage controller, and the support element). Certain system management tasks require the cooperation of firmware components on the central electronic complex (CEC), the cage controller (CC), and the support element (SE). For example, to

0018-8646/04/$5.00 © 2004 IBM

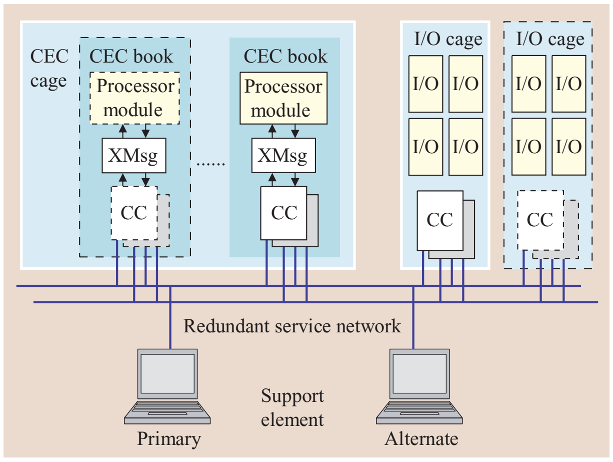

Figure 1

Overview of the z990 system. (CEC: central electronic complex; CC: cage controller.)

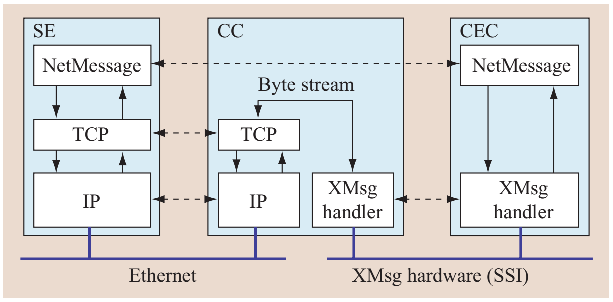

Figure 2

z990 communication infrastructure. (TCP: Transmission Control protocol; IP: Internet protocol; SSI: serial support interface.)

replace an I/O card, it must be removed from the SE view of the system configuration, the firmware running on the processors must be informed that it can no longer access this hardware, and when it can be powered off safely, the CC in the I/O cage where the card resides must be instructed to power off the card. This is just one simple example of a management task, but it shows the need for communication between the involved firmware components. It can be seen in Figure 1 that the support element is connected to the cage controllers via a redundant *service network*, which is realized by Ethernet connections. The cage controllers in the CEC cage are connected to the processor modules via the XMsg-engine[1] hardware in the clock chip. When firmware components residing on the support element have to communicate with firmware components running on the processors, they require the assistance of firmware on the cage controller because there is no direct hardware connection between the support element and the processors.

With the IBM z990 server, the method by which the support element firmware communicates with firmware components running on the processors has been changed. A new protocol has been introduced, and the design, structure, and implementation of the firmware components involved in communication tasks have been changed.

Motivation

Figure 2 shows the new communication infrastructure designed and implemented for the IBM z990 server. First, a short explanation of the new structure is given, and then the rationale for the new design is enumerated.

The support element is connected to the cage controllers via a redundant Ethernet connection, while the cage controllers are connected to the processor module via the XMsg-engine hardware. The standard Transmission Control protocol/Internet protocol (TCP/IP) suite is used for SE communication with the cage controllers. On the side of the CC and processor module, the XMsg-engine handler is used to access the XMsg-engine hardware.

Both TCP and the XMsg-engine handler offer the same logical interface by offering a byte-stream-oriented interface (see [2]). This means that they offer a plain send/receive interface for unformatted data and they guarantee that the data arrives in the same order as sent.

The NetMessage protocol is the protocol for the application level (layers 4–7 in the TCP/IP protocol suite [2]). It is used for communication between the support element firmware components and the firmware components running on the CEC. It defines the format of the data exchanged via TCP/IP and the XMsg engine. While the support element and the CEC firmware use the NetMessage protocol to "understand" each other, the cage controller requires no notion of a protocol. The CC serves only as a bridge/router [2, 3] between the Ethernet and the XMsg-engine hardware. The CC simply has to forward each byte arriving at the TCP protocol level to the XMsg engine. This approach has several advantages: The CC firmware is not affected by any protocol changes in the application layer, no protocol translation has to be done, and so on.

The solid arrows in Figure 2 show the data transmission path, while the dashed-line arrows show the logical communication path on each International Standards

[1] The XMsg engine is a hardware interface on the clock chip that connects the cage controller to the CEC. The XMsg engine contains read/write first-in first-out (FIFO) registers for data exchange as well as several control lines, among them two high-priority reset lines for resetting the communication interface in case of errors. It is connected to the cage controller using the serial support interface (SSI) and is also accessible by millicode.

Organization/Open Systems Interconnect (ISO/OSI) or TCP/IP protocol suite layer.

These changes from our earlier approach were made for the following reasons:

- *Simplification of firmware structures and components.* The new protocol has helped to simplify the structure and the code of firmware components on the SE, CEC, and the CC. Eliminating the need for protocol conversion on the CC is just one example.
- *Elimination of a single point of failure.* The redundant service network and multiple XMsg engines in a multibook system offer the possibility to communicate, even if a hardware link is broken. The firmware components exploit this feature. For example, on the CEC side, the clock chip hardware interface is serviced by a single dedicated processor—the communication master processor—from among the group of system assist processors. The communication master processor is determined at system startup time, but the functionality can be concurrently reassigned to another processor on another book if required in case of hardware failures.
- *Fault tolerance.* With the new structure, an error does not break the whole communication path or terminate applications using it. A reset mechanism was implemented to reset and recover the communication path end to end.
- *Fault isolation/error data collection.* If an error occurs, however, it is necessary to isolate the firmware component where the error occurred and to collect all information needed to analyze the error. This may require information from all firmware components involved in the communication.
- *Simple/enhanced recovery of the whole communication path.* A recovery and reset path was designed and implemented as a parallel path to the "normal" communication path. Using this path, resets can be signaled to the SE, CC, and CEC. This path is used to recover from communication errors.
- *Performance.* The new protocol and firmware structure must be as simple as possible to be fast and efficient.

Off-the-shelf standard protocol software, such as the Remote Procedure Call (RPC) protocol, could not be used because the firmware stack on the CEC side (which consists of millicode and i390 code running on the processors) has to fulfill specific requirements. The millicode layer, which is the lower-level firmware layer, is written in assembler language. It uses a subset of the z/Architecture* instruction set, plus a set of hardware-specific extensions that are executable only in this millicode layer (e.g., instructions not part of the zSeries* architecture to directly access specific hardware resources). The millicode assembler code is unique for each zSeries hardware generation and has to be reworked for each new zSeries system generation. Therefore, standard communication protocol software cannot just be recompiled and run on a millicode level—it would have to be rewritten in this special assembler code and adjusted for each new system.

The i390 part of the communication protocol, which is the higher-level CEC firmware layer and is written in PL8 language, has to fulfill certain requirements that do not allow the use of standard communication protocol code. The i390 kernel does not support multiple processes; instead, for all i390 applications there is a dispatching scheme that expects that they run uninterrupted, but have to adhere to agreed-upon maximum run time per application before they return to the dispatcher. In general, responsiveness is an important issue in CEC firmware, because there are many firmware components that run on the processors and have very tight timing requirements.

Robustness and simplicity are essential in all firmware, but this is especially so for millicode or i390 code firmware, where design or coding errors may bring the whole system down and disrupt the customer's operation.

Also, off-the-shelf middleware usually does not address topics such as redundancy, fault tolerance, and error recovery [4]. Implementing all z990-specific features, such as redundant networks, reassignment of the communication master processor, fault isolation, and error recovery, with off-the-shelf middleware or protocols would have introduced a complex and error-prone firmware structure.

NetMessage protocol

A communication protocol is needed for communication and data exchange between the firmware components residing on the different subsystems—support element and cage controller—and the CEC itself [1, 5]. The protocol must satisfy the requirements of each firmware component. It has to be clear, simple, fast, and easy to implement with different programming languages. The millicode/i390 code implementation must be straightforward, simple, and small in code size. Despite this, the protocol must be easily extensible and general enough to be suitable for future extensions and for firmware components with more processing and memory resources.

The protocol has to provide support for various aspects of firmware communication requirements, such as versioning, unique transaction control, request identification, source/target identification, and error recovery. Thus, compared with a pure transport layer, it provides an additional level of reliability for firmware applications.

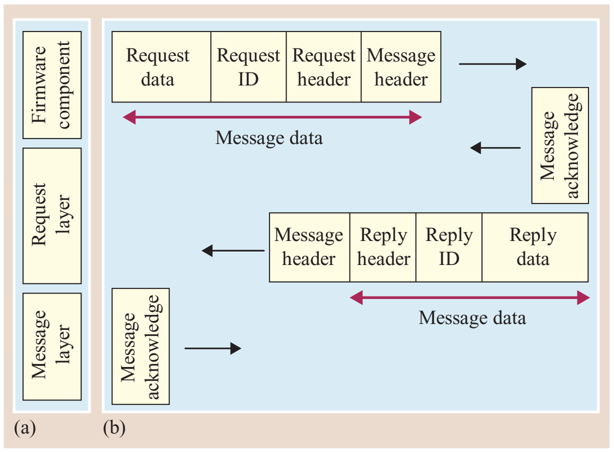

Figure 3

(a) NetMessage protocol layers. (b) NetMessage request layer.

The following areas must be addressed by the protocol:

- Support for bidirectional communication requests.
- Support for multiple parallel communication requests.
- Sending of outbound requests to the correct recipient.
- Routing of inbound requests to the corresponding target application.
- Checking of transactions and appropriate error handling.
- Supporting acknowledgment mechanisms below the transaction level to provide enhanced error handling and fault isolation.
- Recovery from communication and network problems.
- Support for transaction timeouts and appropriate error and recovery handling.
- Support for a reset mechanism to provide stability and reenable communication after the detection and handling of errors.
- First error data capture (FEDC) support.

To satisfy these requirements, the NetMessage protocol was split into two layers, as shown in **Figure 3(a)**. The first layer (message layer) is used for simple message transfer. The second layer (request layer) is used to send messages of different types, such as requests and replies.

Message layer

A message consists of a small header containing administrative data, and the message data itself. The only purpose of the message layer is the safe transfer of messages. Each message is acknowledged by an acknowledge header. The message layer ensures that a message is delivered once (duplicate detection), and recovers in case of errors by retransmitting the same message.

The message layer is able to transfer messages of different message types. The protocol headers provide data elements for detecting and recovering from errors. The protocol handshake provides a way to detect and isolate errors, but the main contributor for fault isolation, recovery, fault tolerance, etc. is the NetMessage message layer implementation. If, for example, the acknowledgment for a message does not arrive in time, it could be that the connection via the Ethernet is broken. The message layer is responsible for detecting this and initiating a reconnect over the second Ethernet. This is just one example of a communication fault.

Request layer

The request layer, shown in **Figure 3(b)**, uses the message layer to transfer different message types (requests and replies, resets, etc.). The request layer implements a simple request/reply protocol. A firmware application on the support element, for example, is able to send a request to another firmware application on the CEC. This request is then executed, and the result is transmitted as a reply back to the support element.

A request includes a request header that contains administrative information, such as the originating application and the application to which the request should be sent. Other parts of a request are a request identification containing information about the kind of request that should be executed and the request data itself. Each request is answered by a reply containing a reply header, a reply type, and the reply data. The message acknowledge headers and message headers in Figure 3(b) are sent by the message layer and are not part of the request layer functionality. The request layer is also able to send other message types for communication management, etc. The capability of sending different message types offers room for future extensions.

Service-word communication using the NetMessage protocol

The support element and CEC applications communicate with each other using *service words.* There are two additional protocol layers required for service-word communication: a service-word handler layer, which consists of a service-word handler both on the support element (CEC communication handler and communication control handler) and on the CEC side (i390 communication handler), and the application layer, which consists of the support element and CEC applications that exchange information using the service-word protocol. The applications on the support element and the CEC are the communication endpoints.

As shown in **Figure 4**, each service word consists of a service-word request (command information and data), and an acknowledgment (service-word reply) from the other side. For simplification, the message acknowledgments from the message layer are omitted in this figure. Space is provided in the service-word header/data structures to support data integrity checking. The entire communication path is time-out controlled.

Each service-word request and reply consists of a service-word request or reply header, respectively, followed by the service-word data to be transported between support element and CEC applications. A service-word request header contains a service-word sequence number, command code, additional fields to further specify the command, information about the logical partition and processor on the CEC that is sending or receiving the command, and information about the length of the application data that follows. A maximum data size of 512 KB was chosen for z990, because it is a good compromise between the requirement to transport large amounts of data within one service word and the memory space that must be reserved on the CEC for buffering service words during the transfer before they are sent to their final destination. However, this size can be adjusted for future projects as necessary. The command code, along with the target processor/logical partition information, is used by the support element and CEC service-word handlers to route the service word to the appropriate support element or CEC application. The service-word sequence number is owned by the application and can be used for recovery, duplicate detection, trace matching, etc.

The acknowledgment from the other side consists of a service-word reply header, which repeats part of the command information and contains an application response code that informs the originator as to the success or failure of the requested operation. Optional reply data can follow the service-word reply header: either application data for good replies or collected FEDC data, which, for error-type replies, aids in finding the root cause of the error.

Both communication endpoints can initiate a service-word communication sequence, as shown in **Figure 5**. Full duplex transfer mode is supported, i.e., an SE-initiated service word and a processing-unit (PU)-initiated service word can be processed simultaneously, but only one service-word transfer initiated by each side is allowed for reasons of service-word handler implementation simplicity. For PU-initiated service-word requests, while a PU-initiated service word is active, a busy-status/application retry mechanism is used (see also the section on i390 implementation). For SE-initiated service-word requests, the SE service-word handler stacks and serializes the

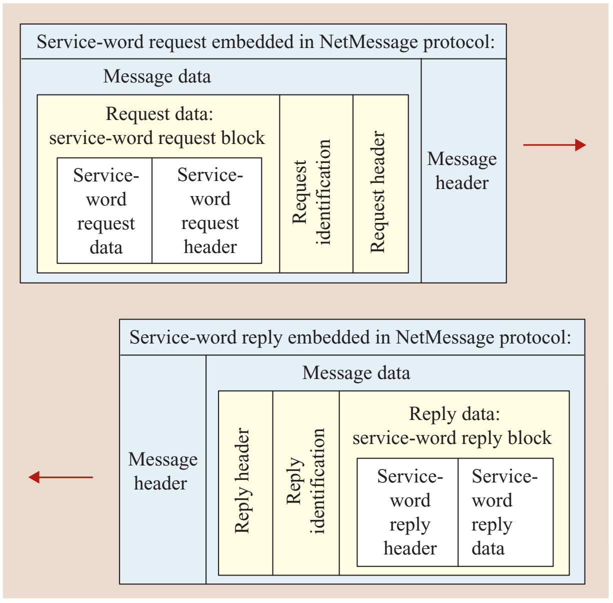
Service-word request embedded in NetMessage protocol:
Message data
Request data: service-word request block
Service-word request data
Service-word request header
Request identification
Request header
Message header
Service-word reply embedded in NetMessage protocol:
Message data
Message header
Reply header
Reply identification
Reply data: service-word reply block
Service-word reply header
Service-word reply data

Figure 4

Embedding service-word request/reply structures in the NetMessage protocol.

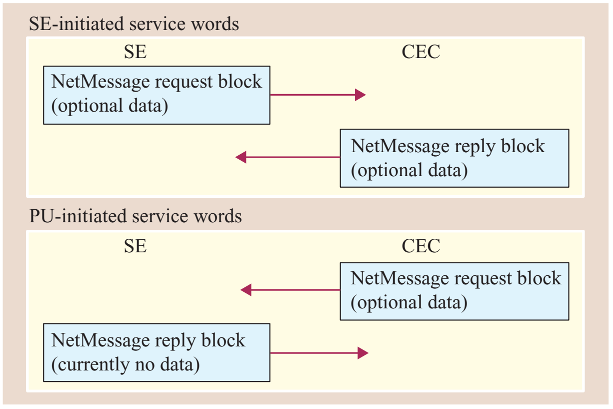
SE-initiated service words
SE
CEC
NetMessage request block (optional data)
NetMessage reply block (optional data)
PU-initiated service words
SE
CEC
NetMessage request block (optional data)
NetMessage reply block (currently no data)

Figure 5

Service-word types.

requests. This means that retry on the application level is not required for SE applications, while the retry on the CEC side is driven by the applications.

There are SE-initiated service words of the `Write` type (up to 512K data bytes can be sent from the SE to the CEC) and of the `Read` type (the SE can request up to 512K data bytes from the CEC). Analogously, the protocol supports both data transfer directions for PU-initiated service words, but currently only the `Write`-type transfer

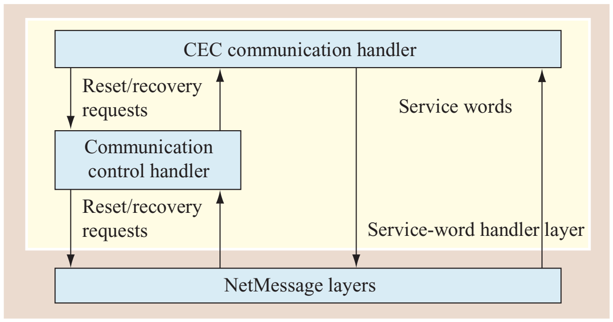
CEC communication handler
Reset/recovery requests
Service words
Communication control handler
Reset/recovery requests
Service-word handler layer
NetMessage layers

Figure 6

Support-element firmware components.

is exploited for these (i.e., CEC sends up to 512K data bytes to the SE, but does not request data from the SE within one service-word communication).

Example of service-word use: Capacity upgrade on demand

A capacity-upgrade scenario for processors (see also [6]) shows a typical usage sequence of service words: The upgrade request is initiated on the hardware master console or SE. After some preprocessing, the SE sends a data record containing the requested processor upgrade configuration to the CEC, using an SE-initiated service word of the `Write` type. The concurrent upgrade application on the CEC then processes the upgrade request and sends a PU-initiated service word that contains (as data) the resulting return code of the upgrade operation to the requesting SE application. In the case of a successful upgrade operation (i.e., there are new processors available), additional PU-initiated service words containing a configuration/state change notification are sent to the SE application that manages processor configuration and state information. This SE application now queries the CEC for current processor configuration and state information using a series of SE-initiated `Read`-type service words. In parallel, the SE concurrent upgrade application has examined the return code of the concurrent upgrade operation on the CEC, and, in case of success, ensures that the logical partition hypervisor and the operating system software are notified that a configuration change has happened. Both notifications are done by using SE-initiated `Write`-type service words.

Once notified, both the logical partition hypervisor and the operating system software can use standard z/Architecture methods to find out about the new configuration and bring the new processors online and into operation. The process of bringing new processors online again results in a series of PU-initiated processor state change notifications and SE-initiated query processor state service words, so that the new processors can be displayed with their correct status on the SE processor view screen.

There are many other typical examples for users of service-word communication in all areas of SE and CEC interaction, either initiated on the SE or by the CEC side. Examples on the SE side include concurrent channel upgrades, concurrent firmware upgrade (patches), system activity display, and channel operations such as configure on/off sequences. On the CEC side, the sending of processor error data and logging information to the SE log file are examples.

Protocol error handling and communication reset sequences

Upon detection of protocol errors (e.g., transaction ID mismatches, cyclic redundancy check errors, and several more) certain other conditions (such as processor reassignment, processor restart in the middle of an ongoing communication, or time-out conditions), both service-word handlers on the SE and CEC can start a reset sequence that will reset the communication path from end to end. Both the hardware and the firmware structures involved in the communication have to be reset to a defined idle state.

An acknowledged reset sequence is used to ensure that the entire communication path is reset. Such a reset may interrupt currently ongoing service-word transfers, and for simplicity and safety of implementation, it was decided to abandon the currently ongoing service-word transfer completely and provide a method for the applications to allow a retry of these aborted service words. The reset sequence is safe, even when both SE and CEC decide to request a reset simultaneously. Since the reset has to have the highest priority in the handling of service words, it is signaled via two high-priority hardware interrupt lines in the XMsg engine, one for each direction.

Support element implementation

The SE firmware components shown in **Figure 6** are responsible for communication with the CEC firmware. They were designed and implemented using object-oriented technologies and well-known design patterns (observer, leader follower, facade, acceptor, connector, etc.) [7].

CEC communication handler

The central part of the SE firmware structure is the CEC communication handler, which provides a communication interface between SE applications and CEC firmware applications. The service-word protocol (see the section above on service-word communication using the NetMessage protocol) is transferred as NetMessage requests/replies. Applications can use the CEC communication handler to send service words to CEC applications and can register for receiving service words from CEC applications. An interface for transferring files, data areas, etc. to the CEC is also provided.

The CEC communication handler passes the service words to the NetMessage layers, where they are packed in a NetMessage request block and sent to the CEC. The NetMessage layer delivers incoming requests containing service words from the CEC side to the CEC communication handler.

In the case of an error, the CEC communication handler uses the communication control handler to query the status of the communication path and to fetch additional error information (FEDC). The communication control handler is also used to reset the entire communication path and to take recovery measures. The communication control handler has registered itself at the NetMessage layer for incoming requests containing recovery/reset requests from the CEC side. It takes the corresponding actions if such a recovery/reset request arrives and instructs the CEC communication handler to perform its recovery/reset actions.

The communication control handler uses the NetMessage layers to contact and communicate with the cage controller. The cage controller is able to read the hardware status of the XMsg engine and from the hardware gathers additional FEDC data that contains more explicit information about the error that occurred. With other words, the communication control handler talks with the firmware residing on the cage controller but not with the firmware on the CEC, while the CEC communication handler talks with the CEC firmware. Depending upon the error information that was gathered, the corresponding recovery actions—e.g., resetting the internal states of the CEC communication handler and resending the service word—are taken.

CEC communication handler virtualization
Since z990 is a multibook system (see also [8]) and consists of up to four books, a CEC communication handler is created for each book in the system. Each CEC communication handler is in contact with the cage controller on the corresponding book. Only one CEC communication handler is active at a time; this is the CEC communication handler for the book in which the communication master processor resides. To simplify access for applications, a virtualization of the CEC communication handler has been introduced. Applications need only know the virtual CEC communication handler. The virtual CEC communication handler redirects incoming and outgoing service words, file transfers, etc., to the currently active CEC communication handler. The active CEC communication handler is changed when the communication master processor is reassigned across books.

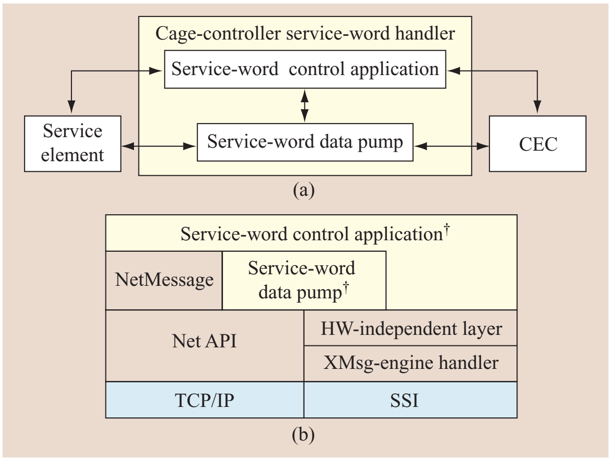

Figure 7

(a) Overview of the z990 cage-controller service-word handler. (b) Cage-controller service-word handler software layer.

z990 cage-controller service-word handler and applications
The standard cage-controller service-word handler is implemented as a bidirectional data pump, forwarding packets to and from the SE and CEC without knowing anything about the protocol. This pump is supervised by a control application to control and synchronize the data pump with the SE/CEC, to collect FEDC data, and to communicate the actual state to the SE/CEC if necessary. **Figures 7(a)** and **7(b)** respectively provide a functional overview and describe the layers of the new cage-controller service-word handler. Applications that are new compared with the z900 are indicated with a dagger symbol. The data pump does not access any NetMessage functions to the TCP/IP network, but directly uses the TCP/IP-based network application programming interface (Net API). To access the XMsg engine, the XMsg-engine handler is used via hardware-independent functions. This is a major advantage because today's applications may now be used in future designs in which the hardware is changed; only a new engine handler is needed, and there will be no change in the behavior of the application. The service-word control application alone communicates via the NetMessage protocol to the SE and, on "the other side," the control application uses the hardware-independent functions to communicate via the XMsg-engine handler to the CEC.

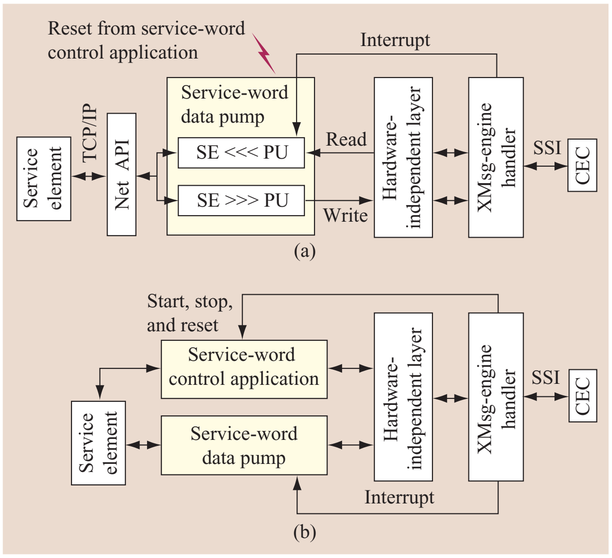

Figure 8

(a) Operation of the service-word data pump. (b) Overview of service-word control application.

Service-word data pump

The control application can start, stop, and reset the data pump [**Figure 8(a)**]. (The data pump forwards the service words from SE to the CEC and vice versa. It is a plain packet forwarder.) Once the data pump is started (the default state), it establishes a Net API (TCP/IP) server connection to the SE. Packets can be received and sent in both directions. There are two independent threads: one to receive packets from the Ethernet and forward them immediately over the serial support interface (SSI)[2] to the CEC, and one to receive packets from the SSI and forward them over the Ethernet to the SE.

The incoming service-word requests/replies are received in small packets to obtain optimum performance. They are immediately copied to an outgoing send buffer. In doing so, the pump does not "look" into a service word and hence does not know anything about its protocol, beginning, or end. To exclude simultaneous access of the threads to the XMsg engine/SSI, the serialization management of the hardware-independent layer is used, which allows only one thread at a time to access the specific hardware. On the Ethernet side, no synchronization is necessary, because send and receive can be performed at the same time on the same socket (handled by TCP/IP). In case of an error, the data pump can also be reset from the service-word control application to ensure that all buffers are empty and the connection to the SE is reestablished.

[2] The SSI connects the clock chip on the CEC with the cage controller.

Service-word control application

To be able to control the service-word data pump, but also to communicate with the SE/CEC for control purposes, it was necessary to introduce the new service-word control application [**Figure 8(b)**]. It starts, stops, or resets the applications. In case of an error, the control application collects all necessary cage-controller FEDC data and sends it to the SE. It performs a reset of the data pump only on request of the SE/CEC. The control application can always be queried to return the actual state of the cage controller/CEC. After a cage-controller reset/reboot, the service-word control application immediately starts the data pump.

FEDC data collection

The main part of the FEDC data is generated by the application traces of the cage controller. All important state changes are written in these trace statements and then sent into the cage controller trace buffers. When triggered by events, the trace buffers are sent by the cage controller or polled by the SE (live traces).

In addition, the service-word control application provides a set of state variables. These can be sent by the cage controller or polled from the SE at any time. Optional flags determine whether the FEDC data should be regarded as a system log entry or just attached to the SE trace. A complete explanation of FEDC data collection may be found in [9].

Millicode implementation

The millicode layer serves as a CEC-resident hardware device driver for the service-word communication. During normal operation, the source or target of the service words on the CEC side is i390 code, but i390 code requires millicode both to send data out and to receive data via the XMsg engine first-in first-out (FIFO) registers. **Figure 9** is an overview of the resources and interfaces used by millicode.

At system initialization time, i390 allocates two service-word buffers, one to handle SE-initiated service words and the other for PU-initiated service words. Using these independent buffers allows full duplex mode: One request can go from CEC to the SE, while the SE itself may send a request to the CEC.

After setup of the buffers, i390 triggers the millicode to initialize the service-word handling. During this initialization, the data FIFO interrupts are enabled and the millicode state is reset.

From then on, the only trigger from i390 code to millicode is to signal the start of a PU-initiated message-block-sending operation as soon as i390 has put the message into a buffer. All other work needed to send out or receive messages up to 512 KB long is directly driven by FIFO interrupts and handled by millicode without i390 intervention.

To send or receive these messages, the XMsg-engine hardware provides programmable interruption logic that allows a filling level to be defined individually—a so-called *watermark*—for the incoming and the outgoing FIFO. The inbound FIFO generates an interrupt when this watermark is exceeded; the outbound FIFO generates the interrupt when the filling level drops below the watermark.

Millicode sets these FIFO-interrupt controls to receive an interrupt whenever the expected amount of data has arrived and can be read from FIFO to be put into memory, e.g., the header length, or when all data provided to the FIFO has been sent out and it is time to refill from memory. The controls are set such that a minimum number of interrupts are generated on the CEC by maximum deployment of the FIFO size. This avoids unnecessary interrupt-created overhead.

After initialization, millicode keeps track of the data flowing in and out. This way millicode always has an expectation about which buffer should be used in which direction and how many bytes are missing until a message ends and a new header should start. Using this information allows some base checking of the incoming data; consistency of the header contents and data integrity and completeness checking are implemented here by millicode, completely transparent for i390 code.

Signaling back from millicode to i390 code is done via events. There are events for completed send operations as well as for completed receive operations to let i390 know that the transfer of a request or reply block has finished. There is also an event for a failed check, such as detection of an inconsistent header or a failed data integrity check. When millicode generates such a millicode-detected error event, it also collects FEDC data which, in this case, consists mainly of the header and/or ending data of the currently processed message. When i390 receives this millicode-detected error event, it catches this FEDC data and prepares to send it to the SE, where it is logged after the following reset sequence is finished. On the basis of this FEDC information, the problem debug is quite efficient in the complex environment of the different code layers that are all playing together in the game.

Besides normal operation, the service-word communication is also used in the system checkstop state to transport error information from the CEC to the SE, called *CECDUMP*. This CECDUMP service-word communication is implemented completely in millicode without i390 code support. The CECDUMP code reuses the functional millicode service-word routines and mechanisms and adds just a small service-word handler on top of them. Reusing this code made the CECDUMP quite reliable because there are fewer special error code paths, but much of the code works and is tested under normal conditions as well.

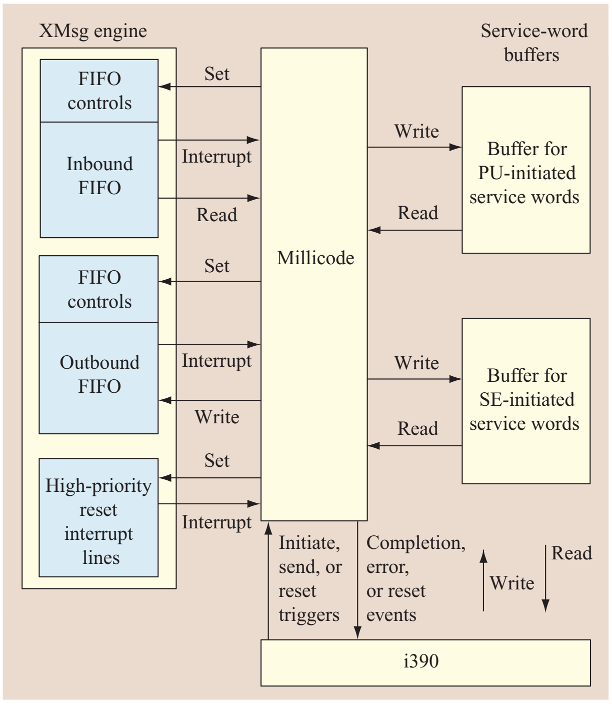
XMsg engine
Service-word buffers
FIFO controls
Inbound FIFO
FIFO controls
Outbound FIFO
High-priority reset interrupt lines
Set
Interrupt
Read
Set
Interrupt
Write
Set
Interrupt
Millicode
Write
Read
Buffer for PU-initiated service words
Write
Read
Buffer for SE-initiated service words
Initiate, send, or reset triggers
Completion, error, or reset events
Write
Read
i390

Figure 9

Millicode overview.

i390 implementation

The i390 and millicode part of the i390 communication handler always runs on one dedicated processor, the communication master processor. To be able to service other processors, the received service-word header contains the target processor number. If the target processor is the communication master processor, the target i390 application is called directly. If a different processor is the target, the status of this processor is checked. The service word can, of course, be distributed only if the target processor is in a state in which it can handle requests from the SE; otherwise, an error reply is returned to the SE.

The target i390 application is identified by a command code in the service-word header. If no receiver application for a certain command code is found, an error reply is sent back.

i390 application interface and handling of PU-initiated and SE-initiated service words

The following are the i390 communication-handler call interfaces with the application:

- The i390 communication handler provides the functions `i390COMM_PUI` and `i390COMM_SPI`, which can be

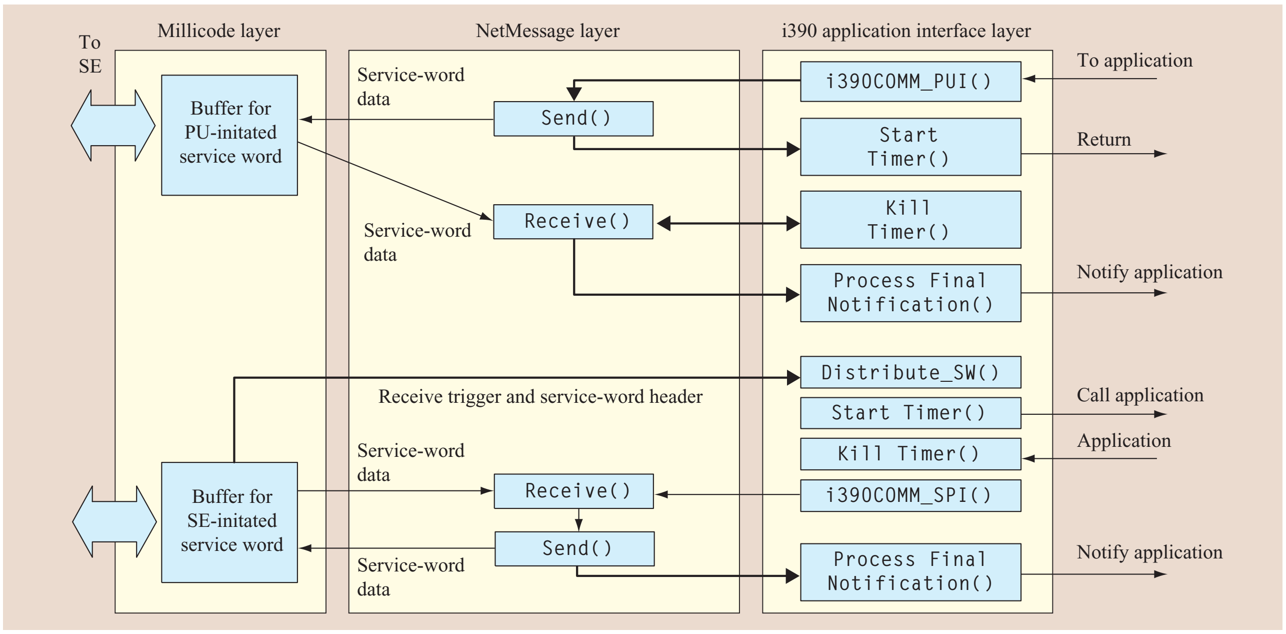

Figure 10

i390/millicode communication flow.

called by the application to handle PU-initiated and SE-initiated service words, respectively.

- Each application that wishes to receive SE-initiated service words has to provide a function for the i390 communication handler to deliver the incoming SE-initiated service words to an *i390 application service-word handler routine*.
- Finally, since the service-word transfer is processed asynchronously, the application has to provide a routine for the i390 communication handler to call when a service-word transfer has finished or to give the application the retry initiative for a previously failed service-word transfer: an *i390 application service-word notification routine*.

Figure 10 shows the communication flow for the two basic communication modes: PU-initiated communication and SE-initiated communication.

PU-initiated communication

Referring to the upper half of Figure 10, it can be seen that the function `i390COMM_PUI` is called from an application to start a PU-initiated service word. In normal operating mode, the i390 communication handler returns control to the application after the request has been given to millicode to send it to the cage controller. The application can then return to the i390 dispatcher, and other i390 work can be processed. Some time later, when millicode notifies the i390 communication handler that the service-word reply has arrived, the application service-word notification routine is called with the final status of the service-word transfer. If the reply does not arrive in time, a time-out condition is raised by the i390 communication handler, and a communication reset sequence is initiated. In this case, the application receives a time-out status as a final transfer status. When the reset sequence is finished, i.e., the SE is responsive again, an optional *recovered from time out* notification is passed to the application service-word notification routine to allow the application to retry the PU-initiated service word.

If the request to send a service word cannot be processed because another PU-initiated service word is currently in process, the return code *i390 communication handler busy* is set for the application upon return from the `i390COMM_PUI` call. When the PU-initiated side of the i390 communication handler is idle again, a *ready after busy* notification is passed to the application, which can then retry the PU-initiated service word.

SE-initiated communication

When an SE-initiated service-word request arrives, the i390 communication handler examines the service-word header to determine the target processor and the target application (see the lower half of Figure 10). If there is no error, the target application is called on the target processor. The i390 communication handler enters a time-out-controlled state waiting for the application to come back with the service-word reply. If there is bad target processor or command code information, appropriate error responses are immediately sent to the SE, as described previously.

If the application returns in time with an `i390COMM_SPI` call and provides a buffer for the request data, the service-word request data already received from the SE is copied to the application data buffer, and the optional i390 application reply data is copied into the millicode buffer to send it to the SE. When the service-word reply, along with optional service-word reply data, has arrived at the SE, a final status notification is generated for the application. If the application has not returned in time with an `i390COMM_SPI` call, the i390 communication handler itself sends an error reply to the SE to finish this service-word transfer according to the protocol rather than provoke a time-out on the SE side.

i390 communication-handler internal states

Because communication from the SE to the CEC and from the CEC to the SE can be active at the same time, the i390 communication handler requires two independent state machines: one for the SE-initiated side and one for the PU-initiated side. They are linked only via the reset path. A reset, which is allowed at any time, causes both state machines to enter the *reset-in-process* state.

Results

A number of improvements were achieved by introducing the NetMessage protocol for CEC-to-SE communication. The following results were observed.

By eliminating the need for protocol conversion on the cage controller, the cage-controller code was significantly simplified, thereby improving its reliability, availability, and serviceability (RAS) characteristics and also the portability of the code. Future protocol changes in the interface between the SE and CEC are transparent to the cage-controller code. Most of the cage-controller implementation is independent of the actual CEC interface hardware. If the CEC interface hardware changes, only the XMsg-engine handler has to be replaced by a new device driver.

The concept of a data pump relieved cage-controller memory requirements dramatically, because there is no longer a need to provide a large buffer for service words.

The number of necessary transfer steps to transfer one service word was reduced from a complex scheme of up to five data-block transfers per service word to only two transfer steps: the NetMessage request and reply block transfers. This simplified the state machines required on the CEC side and created better performance due to less protocol overhead.

Further performance improvements were achieved by increasing the maximum data size from 64 KB to 512 KB per service word. This means less protocol overhead for large data transfers, such as those used for CECDUMP or initial firmware loading. Also, introducing full duplex instead of half duplex mode allows two service words to be active at the same time and reduces service-word handler busy conditions.

Performance comparison of service-word data transfer rates between the z900 and the z990 showed that the performance of diskette IPL ("Load processor from file" function with a sample file size of 8 MB) was improved by more than 50%, and the performance of CECDUMP (sample dump size 40–60 MB) was improved by more than 400%. This performance improvement is, to some extent, achieved through the new protocol described in this paper, but other factors also contribute to this improvement (for example, new, faster hardware on all levels). Since the new protocol is not available on the z900, it is, of course, difficult to say exactly what percentage of the improvement can be attributed to the new protocol.

RAS characteristics have been improved by simplifying the code and by introducing better recovery from error situations and fault tolerance (for instance, the use of acknowledged reset sequences and the introduction of transaction IDs and sequence numbers in the protocol headers). Also, FEDC capabilities have been greatly enhanced; FEDC and continuous trace data is now permanently written into special trace buffers and retrieved and logged when it is required in case of a failure [9].

Looking forward

The future requirements for an SE–CEC communication interface can be easily met by the NetMessage communication concept. For example, the new protocol allows data to be both sent to and received from the other side within one service word.

Also, traditional service-word functionality—such as file transfers, CECDUMP, and logical partition hypervisor communications—can now be realized by exploiting the capabilities of the NetMessage protocol instead of using service words. This reduces the traffic that flows over the service-word path and allows parallel processing of several types of SE-to-CEC communication tasks.

The protocol layering allows adjustments and extensions on each level for all types of future SE-to-CEC communication requirements. New NetMessage message types or request types, etc., can be defined and supported as required.

Summary

By introducing the NetMessage protocol on all interfaces involved in the processor-to-SE communication, z990 offers significant improvements over the z900 and its predecessors in the areas of performance, code simplicity, RAS characteristics, and FEDC concept. Moreover, this flexible, extensible concept for CEC-to-SE communication puts zSeries firmware in a good position for future IBM eServer* projects and their emerging requirements.

Acknowledgments
Development of the new NetMessage protocol service-word interface involved the contributions of many members of firmware development in the IBM Boeblingen, Endicott, and Poughkeepsie laboratories. Among these, the authors especially thank their colleagues Harald Boehm, Wolfgang Fischer, Wolfgang Gellerich, Helga Hermann, Judy S. Johnson, Walter Niklaus, Hartmut Penner, Karlo Petri, Herbert Schein, Manfred Schindler, and Alexandra Winter for their many valuable contributions and creative ideas during the design phase, reviews, simulation (see [10] for z990 simulation enhancements that have been especially helpful for us), bringup, and test of the new NetMessage communication handler. The authors would also like to thank the anonymous reviewers for their many helpful suggestions and review comments.

References
1. F. Baitinger, H. Elfering, G. Kreissig, D. Metz, J. Saalmueller, and F. Scholz, "System Control Structure of the IBM eServer z900," *IBM J. Res. & Dev.* **46,** No. 4/5, 523–535 (July/September 2002).
2. W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley Publishing Co., Inc., Reading, MA, 1994.
3. A. S. Tanenbaum, *Computer Networks*, Third Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1996.
4. P. Jalote, *Fault Tolerance in Distributed Systems*, First Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1994.
5. 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 (July/September 2002).
6. J. Probst, B. D. Valentine, C. Axnix, and K. Kuehl, "Flexible Configuration and Concurrent Upgrade for the IBM eServer z900," *IBM J. Res. & Dev.* **46,** No. 4/5, 551–558 (July/September 2002).
7. D. C. Schmidt, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, John S. Wiley & Sons, Inc., New York, 2000.
8. T. Webel, T. E. Gilbert, and D. Schmunkamp, "Run-Control Migration from Single Book to Multibooks," *IBM J. Res. & Dev.* **48,** No. 3/4, 339–346 (May/July 2004, this issue).
9. S. Koerner, R. Bawidamann, W. Fischer, U. Helmich, D. Klodt, B. K. Tolan, and P. Wojciak, "The z990 First Error Data Capture Concept," *IBM J. Res. & Dev.* **48,** No. 3/4, 557–567 (May/July 2004, this issue).
10. M. Stetter, J. von Buttlar, D. Chan, D. Decker, H. Elfering, P. Gioquindo, T. Hess, S. Koerner, A. Kohler, H. Lindner, K. Petri, and M. Zee, "IBM eServer z990 Improvements in Firmware Simulation," *IBM J. Res. & Dev.* **48,** No. 3/4, 583–594 (May/July 2004, this issue).

Received September 22, 2003; accepted for publication November 24, 2003; Internet publication April 6, 2004

Christine Axnix *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (caxnix@de.ibm.com).* Mrs. Axnix received the Dipl. Ing. degree in electrical engineering from the Berufsakademie, Stuttgart, Germany, in 1989. That same year she joined the IBM Development Laboratories in Boeblingen in the Product Assurance Group, where she worked on S/390 processor and coupling architecture verification test programs. In 1993, she joined the CEC Firmware Development Department, where she has worked on various S/390 and zSeries projects in the i390 code area, including the support element–CEC communication interface and CUoD/CBU. She is currently working on the LIC design and code development in the areas of initial firmware load and concurrent maintenance for the eServer z990 follow-on products.

Eberhard Engler *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (eengler@de.ibm.com).* Mr. Engler received an M.S. degree in physics (Dipl. Phys.) from the University of Tuebingen, Germany, in 1995. In 1996, he joined the IBM Development Laboratories in Boeblingen, working on a handheld data acquisition system. That same year he joined S/390 microcode development, where he is primarily responsible for service-word communication, CECDUMP, and TOD/ETR code.

Stefan Hegewald *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (hes@de.ibm.com).* Mr. Hegewald received the Dipl. Ing. (FH) degree in microsystems and precision engineering from the University of Applied Sciences in Karlsruhe, Germany, in 1981. He joined IBM that same year as a process engineer in the chip manufacturing organization in Sindelfingen, Germany. After retraining in 1986 and 1987, he held a number of positions in design, development, and test of text retrieval and DB/2 text extender. From 1997 to 1999, he contributed to the automotive division with IBM Telematic Solutions. Since 1999, Mr. Hegewald has worked on various S/390 projects, especially the design and implementation of concurrent memory upgrade and concurrent CBU undo. Mr. Hegewald led the overall i390 communication handler development effort. He is currently working on the 9037 Model 2 Sysplex Timer follow-on product.

Thomas Hesmer *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (thomas.hesmer@de.ibm.com).* Mr. Hesmer is a Staff Engineer currently working on zSeries flexible support processor (FSP) applications. He studied electrical, electronic, and automation engineering at the Berufsakademie, Stuttgart, and graduated in 1994, receiving a Diplom-Ingenieur (B.A.) degree. After two years of self-employment and four years of employment as an electronic and software engineer, he joined IBM in 2000 at the Boeblingen Development Laboratory. Since then he has worked on the design and implementation of the service and control network applications for the zServer cage controller. Currently, Mr. Hesmer focuses on the transition of the FSP software from OSOPEN to Linux.

Martin Kuenzel *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (kuenzel@de.ibm.com).* Dr. Kuenzel joined the IBM Boeblingen Laboratories after completing his Ph.D. thesis in solid-state physics at the RWTH, Aachen. Since then he has worked in various areas in the zSeries Support Element Development Department of the IBM Systems Group. As the team leader of the Cage Communication Support Layer Team, he was responsible for the transition to the new CCSL structure on the support element. Since 2003, Dr. Kuenzel has been on international assignment, working for the IBM Endicott Laboratory.

Friedrich Michael Welter *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (fmwelter@de.ibm.com).* Mr. Welter received a Dipl. Inf. (FH) degree in computer science from the University of Applied Sciences, Wuerzburg, Germany, in 2000. He joined IBM that same year, working on the design and implementation of base services for the support element and the design and implementation of simulation components. His work currently focuses on the transition of the support element software to Linux.