1985

A Communication Sub-System for MVS/XA

Dan C. Marinescu

Report Number:
85-555

A COMMUNICATION SUB-SYSTEM FOR MVS/XA

Dan C. Marinescu

CSD-TR-555
October 1985

A Communication Sub-System for MVS/XA

Dan C. Marinescu
Computer Science Department
Purdue University
West Lafayette, IN 47907

Overview

This paper explores inter-process communication based on cross-memory services in MVS/XA and describes the implementation and the use of a simple sub-system, Extended Communication Facility (ECF), offering a number of primitive transport functions (OFFER, CONNECT, SEND, RECEIVE, DISCONNECT) and running in a dedicated address space.

The ideas and mechanisms presented are useful for many scientifical and engineering applications running under MVS/XA. Examples are: communication sub-systems, mail sub-systems, multi-address space sub-systems, applications based upon concurrent programming, a *quasi-batch* environment, etc.

Inter-process communication in MVS/XA

MVS/XA uses the concept of *task* to describe a computation or a dispatchable unit of work, but since the concept of *process* is widely used with essentially the same meaning we shall use the later throughout this paper.

In MVS/XA all user initiated processes, as well as the ones created by the system on the user's behalf are confined to an *address space* which defines the addressing environment for a given user. One recognizes three types of users namely started tasks, which are operator initiated activities, interactive and batch users. Consequently, there are three execution environments, one for each type of address space. Each address space is almost entirely isolated from the others; more precisely there are three areas, a user's private area, a common area and one for system residence, in each address space. The address mapping for the private area is different for each address space while the the mapping for the other two areas is the same for all address spaces. In order to share code or data among address spaces, they should be placed in the common region. There are two obvious problems (and a few non-obvious ones) in this approach:

- the common region becomes overcrowded with code to be shared,
- the ability to share data by placing it in the common are is very limited due to the space constraints and, even more important, the lack of protection makes this solution impractical. There are only sixteen different protection keys, hence, there is no way to limit access to data placed in the common area.

Exchange of messages between address spaces using *standard* MVS techniques is a tedious and expensive activity. Essentially, the sender has to acquire a buffer in the common area, move its data into the buffer, schedule a request via a System Request Block (SRB) for the receiver to pick-up the data and wait until the later posts completion. The very fact that each time a data buffer is exchanged, the system dispatcher must be invoked, gives an idea of the overhead involved in this type of communication and accounts for the CPU usage of the current communication sub-systems, TCAM, VTAM etc. Each of them runs in a dedicated address space as a started task and data buffers are moved to and from the address space where the application runs.

Returning to inter-process communication, we recognize two cases:

-a. Processes which reside in the same address space; communication is done via shared memory. We should point out that the only high level language environment which supports concurrent programming in MVS is PL/I. In PL/I, concurrent processes may be started by call statements which specify one event for each process; events allow process synchronization by using wait and post operations. By default, all user initiated processes run with the same protection key so that private process data cannot be protected. Conceptually it is possible to have different keys associated with different processes running concurrently in a given address space, but it is non trivial to implement this idea and, even more important, there are only eight protection keys in user mode. The lack of protection is probably one of the reasons which has limited the number of applications using concurrent programming in MVS/XA.

-b. Processes which reside in different address spaces: communication can be done either by the *standard* mechanism previously described or by using the cross

memory services (CMS). We have pointed out the drawbacks of the first solution in terms of efficiency and ease of use; though much more efficient, the use of CMS is by no means trivial.

Inter-process communication based on cross-memory services

Dual Address Space (DAS) facility. As far as addressing is concerned, a major enhancement of the 370 architecture can be observed in the 308x and 309x machines [1]. Under the generic name DAS, one recognizes extensions of the PSW (Program Status Word) key, of the storage key, of the program status as well as a set of new machine instructions.

The addressing context of any address space may be switched from a primary to a secondary addressing mode and in this way a program in the private area of one address space may execute code residing in another address space or may move data between the two. Also, code residing in the common area may address data from the private area of any address space by setting it as a secondary address space; then, after switching to the secondary addressing mode, all standard 370 instructions could be executed, to address data in that particular address space.

PSW and storage key extension provides now a list of keys called PKM (PSW Key Mask). The standard fetch and store protection mechanisms still apply but a dispatchable unit of work under a TCB (Task Control Block) or SRB may now easily switch between the keys in PKM. As a side effect, we note that PKM can be used to implement protection for processes running in the same address space.

Extended program status is now maintained in control registers: segment table origin and length for the primary and secondary address space (in CR1 and CR7), linkage table origin and length (in CR5), primary and secondary address space ids (in CR4 and CR3). PSW indicates if in primary or secondary addressing mode.

There are new semi-privileged machine instructions which allow:

- *synchronous transfer of control* from a program in one address space to a program in a different address space (PC-Program Call, PT-Program Transfer),
- *data movement across address space boundary* (MVCP-Move to Primary, MVCS-Move to Secondary, MVCK-Move with Key),
- *data access* (SSAR-Set Secondary As, SAC-Set As Control),
- *key manipulation* (IPK-Insert Program Key, SPKA-Set Psw Key from Address).

Cross Memory Services. CMS provides the software support for dual address space facilities [2],[3]. A set of macro instructions available to *authorized users* (users in supervisor state or keys 0-7) gives table driven authority to establish a cross memory environment and to access data and code across address space boundary.

A dedicated address space, PC/AUTH contains both the code implementing cross memory services and the control blocks for CMS. There are two classes of services provided:

- *authorization services* used to manipulate system wide authorization indexes which determine the level of authority in using cross memory functions.
- *linkage services* which allow *late binding* of a program from one address space to code residing in another one which offers services.

An address space providing services must have an entry in an *Entry Table (ET)*, for each service it offers. This entry contains a description of all address spaces which are allowed to use that service, the virtual address to be given control when the service is invoked, a pointer to a two word parameter list and other informations. Each address space has a *Linkage Table (LT)* which defines all services available to it; an LT entry contains a pointer to the ET origin for the address space offering the service. To establish the cross memory environment, proper entries must be created in each of these tables. At execution time, control is transferred after a two level table look-up, by means of a PC instruction in which the service is identified by a PC number. This number is unique system-wide, consists of an LT index (used to extract the pointer to the ET) and an ET index and leads to the ET entry corresponding to the service. By examining the ET entry the authority to perform the PC can be checked and control can be passed to the proper code.

It is important to note that as far as scheduling is concerned, this transfer of control is done without invoking the system dispatcher and execution continues under the control of the same TCB. This makes CMS efficient and attractive in spite of its complexity.

The use of CMS. Though MVS/XA makes extensive use of CMS by executing specific system functions in dedicated address spaces (GRS-Global Resource Serialization, CONSOLE), the direct use of cross memory services is rather cumbersome due to:

- the difficulty to establish a cross memory environment,
- the large number of restrictions at execution time, with most of the standard system services not available in cross memory mode,
- availability of cross memory services only to authorized users,
- the requirement to make non-swappable the address space which provides services in cross memory mode.

In our opinion a higher level interface, like the one described in this paper, must be provided by the MVS/XA system, to give access to these services to a larger group of users.

Applications of inter-process communication

For many scientific and engineering applications inter-process communication is an important issue. Briefly we shall describe now applications which rely heavily on inter-process communication and which could benefit from the use of CMS.

Networking applications are obvious candidates. A transport station for TCP/IP or other transport protocols not currently supported under MVS/XA, can be designed using two dedicated address spaces. The first one will implement the transport primitive functions of TCP (CONNECT,LISTEN,CLOSE, SEND,RECEIVE) as well as a name server and two other functions, ToNet and FromNet, to communicate with the second address

space where the code implementing the network layer protocol as well as the code for the IMP driver will reside. Since CMS provides an efficient way to move packets of data between the application's address space and the two address spaces dedicated to TCP's transport station, this implementation is adequate.

MVS/XA is an appealing environment for the scientific and engineering community since it allows both interactive and batch processing. Most number cranking applications run in batch mode and the user has no control upon his job until it is finished. Sometimes a significant amount of computing resources is wasted since in a long computation, incorrect results occur in an early phase. It would be ideal for large batch programs to have check-out points at which to report partial results to an interactive user and to be able to carry on the remaining computation only after a dialogue in which computational parameters could be adjusted. Such a *quasi-batch* environment can be easily implemented using inter-process communication primitive functions as the ones provided by ECF.

There are applications which may be best implemented as *multi-address space systems*. Let us consider a community of users performing real-time data acquisition and analysis. For each user the environment consists of a number of processes, one say prR, to read input data, perform some sort of data compression and write it on some output media, a second one, say prA, to take samples of input data and run them through an analysis procedure, a third one, say prC, to read, interpret and execute input commands, a fourth process, say prL, to present in a *life graphics* mode the results of the analysis, etc. All these processes run concurrently, act upon a different stream of data and have adequate priorities; prR must have a high priority so that all input data can be captured, prA should have a low priority and process a variable percentage of input data, depending upon the input data rate, the overall system load etc., prC should have the highest priority to allow user control. If all processes share the same address space, elaborate mechanisms must be built-in to ensure: serialization of non-sharable resources, protection, process synchronization, etc. Though most of the code can be carefully designed and checked, the analysis procedure, which is user specific, it is very often not properly debugged and may be the Trojan horse in the environment. For this reason a good solution is to split such an application, to have an address space, say asM, where the standard code runs and a number of satellite address spaces asA1, asA2, ..etc. where analysis procedures reside.

Inter-process communication using ECF

Extended Communication Facility (ECF) provides a high level interface to cross memory services and allows inter-process communication across address space boundaries for all users running under MVS/XA.

Any ECF transport function is initiated from the user's application layer. An active user, say A, wishing to communicate, issues an OFFER, making himself known to ECF. Any other user wishing to communicate with A, may establish a communication path to it, by issuing a CONNECT request for A. After a successful connect, the requester obtains a *token* which uniquely identifies its partner; a *token* is a logical name of a communication partner which may be either a TSO user (in this case the user identification

must be specified in the connect request), a batch job (the job name must be given) or a started task. Subsequently, each partner may SEND or RECEIVE messages from any other address space connected to it until one of the partners performs a DISCONNECT.

In the present implementation, up to 170 address spaces may use ECF in the same time and any of them may be connected up to 50 others, at any time. ECF provides asynchronous communication and allows up to 10 messages to be queued for any destination address space from each of its partners. A message may have any length up to 32 kbyte. All these parameters can be easily changed with more appropriate ones at the time when ECF is installed, depending upon the application; for example a graphics application may require messages of a few Mbyte. Communication with ECF if fully transparent, the message may contain data, code, control information, etc.

The same user interface may be used either from PL/I or from FORTRAN environments. Each primitive function returns a completion code and additional information; for example a SEND returns the number of unread messages in partner's *inbox*, a RECEIVE returns the number of messages in owner's *inbox*, a special form of RECEIVE provides a lis9t of all partners connected to the owner and the number of messages received from each of them.

A very simple *name server* is embedded into ECF. It maps the TSO user identification or the batch job name into the address space identification (ASID) and uses this as a *token*. A slight inconvenience results from this approach; one may communicate only with users which are currently active in the system. Whenever an user leaves the system and eventually comes back, the communication path must be reestablished.

Preliminary measurements indicate that ECF is rather efficient. With a 3081 model D, a SEND and a RECEIVE, from PL/I environment need 6.3 msec for a 32 kbytes message and 2.5 msec for a 4 kbytes message, all overhead included. This amounts to a transfer data rate from one address space to another of about 5 Mbyte/sec for 32 kbyte messages and 1.6 Mbyte/sec for 4 kbytes messages. Since there is a fixed amount of overhead associated with each data transfer and since the data transfer time is much smaller than this overhead, the larger the message is, the higher is the overall data rate attainable with ECF or with any other cross memory based method. A reasonable optimization effort could probably decrease this fixed overhead and lead to higher transfer rates.

We cannot confirm that intensive use of cross memory services in a user application may have a negative impact upon overall system performance as stated in [2] and [3]. Even when the system was heavily loaded, no adjustment of the standard tuning mechanism was necessary during our tests and no increase in the average response time or system overhead was noticeable.

As far as the structure of ECF is concerned one recognizes a monitor program, ECFMON, which runs as a started task in its own address space, ECFAS and the set of primitive transport functions (OFFER, CONNECT, SEND, RECEIVE, DISCON) providing the user's interface; each of these functions performs a PC (Program Call) to the corresponding functional routine in ECFMON. The monitor (Figure 1) consists of:

- a nucleus, ECFNUC, which provides the operator communication interface, is used to build-up the cross memory environment, does error handling, etc.
- a set of functional routines, ECFPC1,ECFPC2,..ECFPC5, which perform the services offered by ECF,

- a set of auxiliary procedures.

ECF user interface

To use ECF, an application layer must be designed using ECF's transport primitive functions, available via standard subroutine call in PL/I or FORTRAN environment.

The object modules of the five primitive functions may be part of user's private libraries, provided that the installation has an SVC to bypass the APF authorization; otherwise the user load modules must be catalogued in an authorized library. The first solution has distinct advantages.

The user's load module should include all primitive function code. The storage requirements for them are minimal, in the 2 kbytes range. The user should provide a buffer area equal in size to the maximum message length, for receiving messages.

The application layer for ECF communication. An user may enter the ECF subsystem, either by performing an OFFER, or by performing a CONNECT to another user already active in ECF. After a successful OFFER or CONNECT, the user gets an entry in the Table of Active users, TA, and a Table of Connected users, TC, is allocated to him. These two elements, the TA entry and the TC table, exist until the owner leaves ECF, by performing a DISCONNECT. All messages are held in a double-chained queue pointed at, from the sender's entry in the TC owned by the user to whom the message is sent.

The full description of ECF primitive transport functions, as well as the significance of all parameters is presented in Appendix 1. The error codes are self-explaining and the following examples describe the actions taken in each case.

It is the user's responsibility to check the return codes after each call and to take proper actions; for example in case of a return code of 2, a delay of a few milliseconds and a retry is indicated. The use of ECF functions is serialized, any user must obtain the CML lock of the address space where ECFMON runs; whenever this lock is unavailable the requester must retry later.

The first example presented in Appendix 3 illustrates the design of a service subsystem, using ECF. A batch job which performs a specific service for all requesting address spaces uses ECF to receive the service request, interprets it, performs the service and provides the results after service completion. Essentially, the server polls its TC and whenever it finds a message reads it. In our example the server checks every ten seconds the number of messages received from all connected users and echos the message. Such a server may be a magnetic tape handling sub-system, a graphics sub-system to handle non-interactive graphic devices, etc. As a general rule one may design such a server sub-system whenever in the interactive environment a number of users need a service which may be performed asynchronously. This approach has the advantage that it prevents code duplication.

The procedure presented in the second example of Appendix 3 illustrates an application layer for a dialogue oriented environment and it was actually used to test different ECF functions. The error processing is sketchy in order to preserve the comprehensibility of this example. Basically, the program prompts for the desired ECF function and its required parameters and executes the function.

ECFMON structure and implementation

ECFMON runs as a started task in its own address space, ECFAS, and consists of a nucleus, ECFNUC, a set of functional routines which perform the services offered by ECF and a set of auxiliary functions. The structure of ECFMON and the map of ECFAS is presented in Figure 1. The current version of ECFMON consists of less then 4 kbytes of code. Nevertheless, in order to have enough space for data buffering, it is advisable to have for ECFAS an address space of large size.

ECF nucleus. The nucleus, ECFNUC performs a variety of functions necessary to create the cross memory environment, to built the control structures of ECF, to provide operator communication interface.

When started by operator, ECFMON must first take actions to set its address space, ECFAS, as a privileged one. Since cross memory services are available only to requesters in *supervisor state and PSW key 0 to 7*, a MODESET macro instruction must be performed. In order to have the authorization to use this macro instruction, the ECFMON must be linked with an Authorization Code (AC) of one, in an APF authorized library. Another way is to design an SVC to bypass the authorization mechanism; basically this SVC must be able to turn ON and OFF the JSCBAUTH bit in the JSCB (Job Step Control Block), which is tested by TESTAUTH macro before performing any restricted services. Though ECFMON may be catalogued in a system APF authorized library, this SVC is important since the primitive functions of ECF must also run in the same privileged mode as ECFMON and it is not advisable to catalog the user's load modules in a privileged library, but to use this SVC. Another requirement is to have the ECFAS as a non-swappable address space and this is done by means of a SYSEVENT DONTSWAP macro instruction.

Now the cross memory services may be used to perform the following actions: set a system-wide linkage structure by using LXRES macro to reserve a linkage index and set the authorization index to one by means of an AXSET macro. An Entry Table for ECF is then defined (by using the ETDEF macro) and the PC numbers for the services provided by ECF are stored in a communication vector table. To conclude the building up of the cross memory environment, the services provided by ECF must be connected using a ETCRE macro.

The next step is to construct ECFMON's own control structures. The first one is the system's Table of Active users (TA). The format of this table is described in Appendix 2. Next, the ECF communication area is built. Since this area must be accessible to all users, it should reside in the Common Storage Area (CSA) and must be pointed at, from the Common Vector Table (CVT). Once the space for the communication area is allocated, the pointer to it must be placed in the user field of CVT. Space allocation in CSA proved to be a daring experience for us. One of the global system locks (SALLOC) must be obtained in order to allocate space in CSA and since a branch entry to the GETMAIN macro must be used, special care is necessary in handling the save areas. We have experienced one error when even the GTF trace could provide very little help.

Another function of the nucleus is to provide an operator communication interface. This interface should be able to process MODIFY and STOP commands. In case of a STOP command the cross memory environment must be distroyed.

The functional routines of ECFMON. The five functional routines perform the functions provided by ECF and they are activated via PC's (Program Calls) from the user' interface procedures. Standard PCLINK (STACK and UNSTACK) macros are used to maintain program call linkage information.

For efficiency reasons all parameters are passed to the PC routines via registers (R0, R1, R15); when returning, R1 contains the return code.

Addressability in the PC routines, after space switching is provided via the ECF communication area where the base registers are saved.

Conclusions

It is highly desirable to make available all system services provided by an operating system in the environment of all high level languages supported and not only in the assembler language environment as it is now the case with MVS/XA.

Cross memory services are an extreme example, even for MVS/XA, as far as the difficulties to use it are concerned. Nevertheless, with a relatively modest effort of about half a man-year a high level interface for cross memory services was designed. Inter-process communication based on CMS is much more efficient than the standard MVS method and ECF opens this area for a large class of applications.

References

[1] IBM System/370 Principles of Operation, IBM Publication GA22-7000-8

[2] OS/VS2 System Programming Library: Supervisor, IBM Publication GC28-1046

[3] U. Pimisken and P.Dorn 'Cross Memory Services User's Guide', Technical Bulletin, IBM Washington Systems Center, GG22-9231-00

APPENDIX 1 - ECF primitive functions

Name: OFFER

Function: enter the ECF sub-system

Call: CALL OFFER(RETURNCODE)

RETURNCODE is an output parameter of type BIN FIXED(31)
- 0 - successful offer request;
- 1 - user already in ECF;
- 2 - ECFMON is now active for another user, try later;
- 6 - ECFMON is not yet started;
- 10 - internal ECF error; failure to allocate TC of requester;
- 11 - maximum number of ECF active users reached (170);

Name: CONNECT

Function: connect current ECF user to the one specified in the request.

Note: if successful, TOKEN, will contain identification to be used in all SEND or RECEIVE to or from that partner.

Call: CALL CONNECT (NAME, TOKEN, RETURNCODE)

NAME is an input parameter of type CHAR(8) containing
the userid - if TSO user or,
the job name - if batch job or,
the name - if started task.
It identifies the communication partner.

TOKEN is an output parameter of type BIN FIXED(31) returning the ECFID of partner with given NAME, if successful connect.

RETURNCODE is an output parameter of type BIN FIXED(31)
- 0 - successful connect request;
- 1 - connection already established;
- 2 - ECFMON is now active for another user, try later;
- 3 - partner (NAME) is not active in ECF;
- 4 - invalid NAME supplied;
- 5 - partner (NAME) is not active in MVS;
- 6 - ECFMON is not yet started;
- 7 - successful reconnect;
- 10 - internal ECF error; failure to allocate TC of requester;

11 - maximum number of ECF active users reached (170)
22 - maximum number of partners connected to requester reached (50);
30 - maximum number of partners connected to NAME reached (50);

Name: DISCONNECT

Function: leave ECF.

Note: there are two modes of the disconnect request:
conditional - the requester has:
- all messages sent by all his partners deleted,
- its TC is freed and its TA entry is deleted,
- all messages sent by the requester to others are kept

unconditional - in addition to previous actions, all messages sent to others are deleted.

Call: CALL DISCON (MODE, RETURNCODE)

MODE is an input parameter of type BIN FIXED(31) which defines the type of the disconnect request:
0 - conditional
1 - unconditional

TOKEN is an output parameter of type BIN FIXED(31) returning the ECFID of partner with given NAME, if successful connect.

RETURNCODE is an output parameter of type BIN FIXED(31)
0 - successful disconnect request;
2 - ECFMON is now active for another user, try later;
3 - requester is not active in ECF;
6 - ECFMON is not yet started;
10 - internal ECF error; failure to deallocate the message box;
11 - internal ECF error; failure to deallocate the TC;

Name: SEND

Function: send a message to a specified ECF partner.

Call: CALL SEND (TOKEN, MSGADR, MSGLEN, NMESGS, RETURNCODE)

TOKEN is an input parameter of type BIN FIXED(31) containing the ECFID of partner previously obtained in a CONNECT request.

MSGADR is an input parameter of type pointer; points to the message area in user's address space.

MSGLEN is an input parameter of type BIN FIXED(31); it defines the true message length.

NMESGS is an output parameter of type BIN FIXED(31); it returns the number of messages sent and not read to the partner.

RETURNCODE is an output parameter of type BIN FIXED(31)
- 0 - message successfully sent;
- 1 - maximum number of messages in the inbox (10) reached;
- 2 - ECFMON is now active for another user, try later;
- 3 - partner (TOKEN) is not active in ECF;
- 4 - partner (TOKEN) and requester are not connected;
- 6 - ECFMON is not yet started;
- 7 - message not sent, invalid TOKEN;
- 8 - message not sent, invalid message address;
- 9 - message not sent, invalid message length;
- 10 - internal ECF error; failure to allocate the message box;
- 11 - internal ECF error; invalid next message pointer;

Name: RECEIVE

Function: receive a message from a specified ECF partner.

Note: a list of all partners connected to the requester and the number of messages received from each of them is returned when TOKEN is specified as '*'.

Call: CALL RECEIVE (TOKEN, MSGADR, MSGLEN, NMESGS, RETURNCODE)

TOKEN is an input parameter of type BIN FIXED(31) containing the ECFID of partner; previously obtained in a CONNECT request.

MSGADR is an input parameter of type pointer; points to the message area in user's address space where the message will be found; this area should be of the maximum message size, now 32 kbyte;

MSGLEN is an output parameter of type BIN FIXED(31); the true message length will be returned here;

NMESGS is an output parameter of type BIN FIXED(31); it returns the number of messages in the inbox of the owner;

RETURNCODE is an output parameter of type BIN FIXED(31)
- 0 - message successfully received;
- 1 - no message from TOKEN in the inbox;

2 - ECFMON is now active for another user, try later;
3 - partner (TOKEN) is not active in ECF;
4 - partner (TOKEN) and requester are not connected;
6 - ECFMON is not yet started;
7 - message not received, invalid TOKEN;
8 - message not received, invalid message address;
10 - internal ECF error; failure to deallocate the message box;

APPENDIX 2 - Data structures and auxiliary routines of ECFMON

The following auxilliary routines are called via branch entries from the functional routines:

PCINIT	-	It initializes for a PC by performing a PCLINK STACK;
PCEXIT	-	Exit from a PC routine by performing a PCLINK UNSTACK;
TAADD	-	Processor for the OFFER PC routine;
CONECT	-	Processor for the CONNECT PC routine;
TSRCH	-	Search a table for a given token (ASID);
TCADD	-	Add an entry in the TC of requester;
MSSEND	-	Processor for the SEND PC routine;
MSRECV	-	Processor for the RECEIVE PC routine;
DISCON	-	Processor for the DISCONNECT PC routine;
TADEL	-	Delete an entry from TA;
TCDEL	-	Delete an entry from TC;
DELMSG	-	Purge the message queue of a given user;

The following control structures are used in ECFMON:

Name: ECFCOM;
Function: ECF Communication Area;
Location: In CSA (Common Storage Area), subpool 231;
Pointed from: The user field in CVT;
Length: 52 byte;
Remarks: It is allocated when ECF is started;
Structure:

+0	LXCOUNT	-	Number of LX requested;
+4	LXVALUE	-	LX returned by LXRES macro;
+8	AXCOUNT	-	Number of AX requested;
+10	AXVALUE	-	AX returned by AXRES macro;
+12	TKCOUNT	-	Number ETS created;
+16	TKVALUE	-	Token returned by ETCRE macro;
+20	SERV1PC	-	The PC number of the first service (OFFER);
+24	SERV2PC	-	The PC number of the second service (CONNECT);
+28	SERV3PC	-	The PC number of the third service (SEND);
+32	SERV4PC	-	The PC number of the fourth service (RECEIVE);
+36	SERV5PC	-	The PC number of the fifth service (DISCONNECT);
+40	SCTAPTR	-	Pointer to the TA (Table of Active Users);
+44	SCBASER	-	Base register for ECFMON;
+48	ECFASCB	-	Pointer to ASCB of ECFAS;

Name: TA;
Function: Table of Active ECF users. There is a unique TA in the system;
Location: In ECFAS, in user's private area, in subpool 2;
Pointed from: ECFCOM;
Length: 1024 bytes = 2 + 2 + (170 entries * 6 bytes/entry) ;
Remarks: It is allocated when ECFMON is started; an entry has 6 bytes;
Structure:

+0	NRACTTA	-	Number of active TA entries;
+2	MAXNRTA	-	Maximum number of TA entries (now 170);
+4	TAASID1	-	First entry, ASID of owner;
+6	TATCPTR	-	First entry, pointer to TC of owner;
+10	TAASID2	-	Second entry, ASID of owner;
+12	TATCPTR	-	Second entry, pointer to TC of owner;

.....

Name: TC;
Function: Table of Connected users. There is one TC for each ECF user;
Location: In ECFAS, in user's private area, in subpool 4;
Pointed from: the TA entry of the owner;
Length: 704 bytes = 2 + 2 + (50 entries * 14 bytes/entry) ;
Remarks: It is allocated when the first connect request for the owner is made and it is deallocated when the owner performs a disconnect.
An entry is 14 bytes long.
Structure:

+0	NRACTTC	-	Number of active TC entries;
+2	MAXNRTC	-	Maximum number of TC entries (now 50);
+4	TCASID1	-	First entry, ASID of first partner;
+6	TCFLAG1	-	First entry, flags;
+8	TCEINCN	-	First entry, 'in' message count;
+9	TCEINPT	-	First entry, pointer to the 'inbox';
+13	TCOUTCN	-	First entry, 'out' message count;
+14	TCOUTPT	-	First entry, pointer to the 'outbox';
+18	TCASID2	-	Second entry, ASID of second partner;

.....

APPENDIX 3 - Examples of an application layer for ECF communication

EXAMPLE 1 - A service sub-system

```
TESTBAT: PROC OPTIONS(MAIN) REORDER;
 /* The declarations are the same as in the previous example */
   CALL OFFER(RETCODE);
G_BEGIN: DELAY (10000);
   TOKEN = UNSPEC('*');
   CALL RECEIVE(TOKEN,P_BUFF,MSGLEN,NMESGS,RETCODE);
   MESSAGE=SUBSTR(C_BUFF,1,MSGLEN);
   KSTART = 0;
   IF RETCODE=0 THEN DO;
           G_LOOP: IF KSTART ¬< MSGLEN THEN GO TO G_BEGIN;
           ECFIDS= UNSPEC(SUBSTR(MESSAGE,KSTART+1,2));
           INMSG= UNSPEC(SUBSTR(MESSAGE,KSTART+3,1));
           IF INMSG ¬=0 THEN DO;
             ECFID=ECFIDS;
             GO TO G_RECEIVE;
           END;
           KSTART = KSTART+3;
           GO TO G_LOOP;
    END;
    IF RETCODE¬=0 THEN  GO TO G_BEGIN;
G_RECEIVE:  CALL RECEIVE(ECFID,P_BUFF,MSGLEN,NMESGS,RETCODE);
    IF RETCODE=0 THEN DO;
         MESSAGE=SUBSTR(C_BUFF,1,MSGLEN);
         IF SUBSTR(MESSAGE,1,6) = 'ENOUGH' THEN GO TO G_TERM;
         IF SUBSTR(MESSAGE,1,10)= 'DISCON' THEN GO TO G_DISCON;
         C_BUFF='I AM SNOPPY AND I HAVE GOT YOUR MESSAGE << '||
                                  MESSAGE || ' >> GO ON **';
         MSGLEN=MSGLEN+56 ;
         CALL SEND(ECFID,P_BUFF,MSGLEN,NMESGS,RETCODE);
         GO TO G_BEGIN;
    END;
    IF RETCODE ¬=0 THEN  GO TO G_BEGIN;
G_DISCON: CALL DISCON(MODE,RETCODE);
G_TERM:
END TESTBAT;
```

EXAMPLE 2 - A dialogue oriented application of ECF

```
ECFT: PROC OPTIONS(MAIN) REORDER;
 DCL PLIXOPT CHAR(40) VAR INIT('NOSPIE NOSTAE') STATIC EXTERNAL;
 DCL OFFER ENTRY(BIN FIXED(31)),
     CONNECT ENTRY(CHAR(*) VAR,BIN FIXED(31), BIN FIXED(31)),
     SEND ENTRY(BIN FIXED(31),POINTER,BIN FIXED(31),
                BIN FIXED(31),BIN FIXED(31)),
     RECEIVE ENTRY(BIN FIXED(31),POINTER,BIN FIXED(31),
                BIN FIXED(31),BIN FIXED(31)),
     DISCON ENTRY(BIN FIXED(31),BIN FIXED(31));
 DCL (ADDR,LENGTH,SUBSTR,INDEX,UNSPEC) BUILTIN,
     (SYSIN,SYSPRINT) FILE;
 DCL C_ANSWER CHAR(80) VAR INIT(' '),C_ANS CHAR(1) INIT(' '),
     C_BUFF CHAR(32767) INIT(' '),MESSAGE CHAR(32767) VAR,
     C_BLNK CHAR(4) INIT(' '),P_BUFF POINTER INIT(ADDR(C_BUFF));
 DCL (MSGLEN,NMESGS,RETCODE,ILEN,I,ECFID,TOKEN,MODE,KSTART)
     BIN FIXED(31) INIT(0),(ECFIDS,INMSG) BIN FIXED(15) INIT(0);
 DCL NAME CHAR(8) VAR INIT(' ');
G_BEGIN:
   PUT SKIP EDIT ('ENTER OPERATION CODE (O,C,S,R,L,D,E)') (A);
   GET EDIT (C_ANSWER) (A(80));
   C_ANS = SUBSTR(C_ANSWER,1,1);
   UNSPEC(C_ANS) =  UNSPEC(C_ANS) |'01000000'B;
   SELECT (C_ANS);
     WHEN('O') GO TO G_OFFER;
     WHEN('C') GO TO G_CONNECT;
     WHEN('S') GO TO G_SEND;
     WHEN('R') GO TO G_RECEIVE;
     WHEN('L') GO TO G_LIST;
     WHEN('D') GO TO G_DISCON;
     WHEN('E') GO TO G_TERM;
     OTHERWISE GO TO G_BEGIN;
   END;
G_OFFER:  CALL OFFER(RETCODE);
   SELECT (RETCODE);
     WHEN(0) PUT SKIP EDIT ('WELCOME TO ECF. YOU ARE IN NOW ') (A);
     OTHERWISE PUT SKIP EDIT ('OFFER ERROR'||RETCODE) (A);
   END;
   GO TO G_BEGIN;
G_CONNECT:
   PUT SKIP EDIT ('ENTER USERID/JOBNAME OF PARTNER') (A);
   GET EDIT (NAME) (A(8));
   ILEN=LENGTH(NAME);
   IF ILEN < 4 THEN GO TO G_CONNECT;
   DO I=1 TO ILEN;
       C_ANS = SUBSTR(NAME,I,1);
       UNSPEC(C_ANS) =  UNSPEC(C_ANS) |'01000000'B;
       SUBSTR(NAME,I,1) = C_ANS;
   END;
   CALL CONNECT(NAME,ECFID,RETCODE);
   SELECT (RETCODE);
```

```
      WHEN(0) PUT SKIP EDIT ('CONNECT TO:'||ECFID||' DONE') (A);
      OTHERWISE PUT SKIP EDIT ('CONNECT ERROR'||RETCODE) (A);
   END;
   GO TO G_BEGIN;
G_SEND:
   PUT SKIP EDIT ('ENTER TEXT TO BE SENT TO ECFID:'||ECFID) (A);
   GET EDIT (C_BUFF) (A(32767));
   MSGLEN=INDEX(C_BUFF,C_BLNK)-1;
   CALL SEND(ECFID,P_BUFF,MSGLEN,NMESGS,RETCODE);
   SELECT (RETCODE);
      WHEN(0)   PUT SKIP EDIT ('SUCCESFULL SEND TO:'||ECFID) (A);
      OTHERWISE PUT SKIP EDIT ('SEND ERROR:'||RETCODE) (A);
   END;
   GO TO G_BEGIN;
G_LIST: TOKEN = UNSPEC('*');
   CALL RECEIVE(TOKEN,P_BUFF,MSGLEN,NMESGS,RETCODE);
   KSTART = 0;
   MESSAGE=SUBSTR(C_BUFF,1,MSGLEN);
   SELECT (RETCODE);
      WHEN(0) DO;
        PUT SKIP EDIT ('# OF PARTNERS CONNECTED:'||NMESGS) (A);
        G_LOOP: IF KSTART ¬< MSGLEN THEN GO TO G_BEGIN;
        ECFIDS= UNSPEC(SUBSTR(MESSAGE,KSTART+1,2));
        INMSG= UNSPEC(SUBSTR(MESSAGE,KSTART+3,1));
        PUT SKIP EDIT ('ECFID:'||ECFIDS||'# OF MESSAGES:'||INMSG) (A);
        KSTART = KSTART+3;
        GO TO G_LOOP;
      END;
      OTHERWISE PUT SKIP EDIT ('LIST ERROR:'||RETCODE) (A);
   END;
   GO TO G_BEGIN;
G_RECEIVE:  CALL RECEIVE(ECFID,P_BUFF,MSGLEN,NMESGS,RETCODE);
   SELECT (RETCODE);
      WHEN(0) DO;
          MESSAGE=SUBSTR(C_BUFF,1,MSGLEN);
          PUT SKIP EDIT ('MESSAGE CONTENT: '||MESSAGE ) (A);
          PUT SKIP EDIT ('MESSAGE LENGTH IS:'||MSGLEN ) (A);
          PUT SKIP EDIT ('NUMBER OF MESSAGES:'||NMESGS ) (A);
      END;
      WHEN(1) PUT SKIP EDIT ('NO MESSAGE IN INBOX') (A);
      OTHERWISE PUT SKIP EDIT ('RECEIVE ERROR:'||RETCODE) (A);
   END;
   GO TO G_BEGIN;
G_DISCON:  PUT SKIP EDIT ('ENTER MODE, C(COND)/U(UNCOND)') (A);
   GET EDIT (C_ANSWER) (A(80));
   C_ANS = SUBSTR(C_ANSWER,1,1);
   UNSPEC(C_ANS) =  UNSPEC(C_ANS) |'01000000'B;
   SELECT (C_ANS);
      WHEN('C') MODE=0;
      WHEN('D') MODE=1;
      OTHERWISE GO TO G_DISCON;
   END;
   CALL DISCON(MODE,RETCODE);
   SELECT (RETCODE);
      WHEN(0) PUT SKIP EDIT ('SUCCESFULL DISCONNECT') (A);
      OTHERWISE PUT SKIP EDIT ('DISCONNECT ERROR:'||RETCODE) (A);
   END;
   GO TO G_BEGIN;
G_TERM:  END ECFT;
```

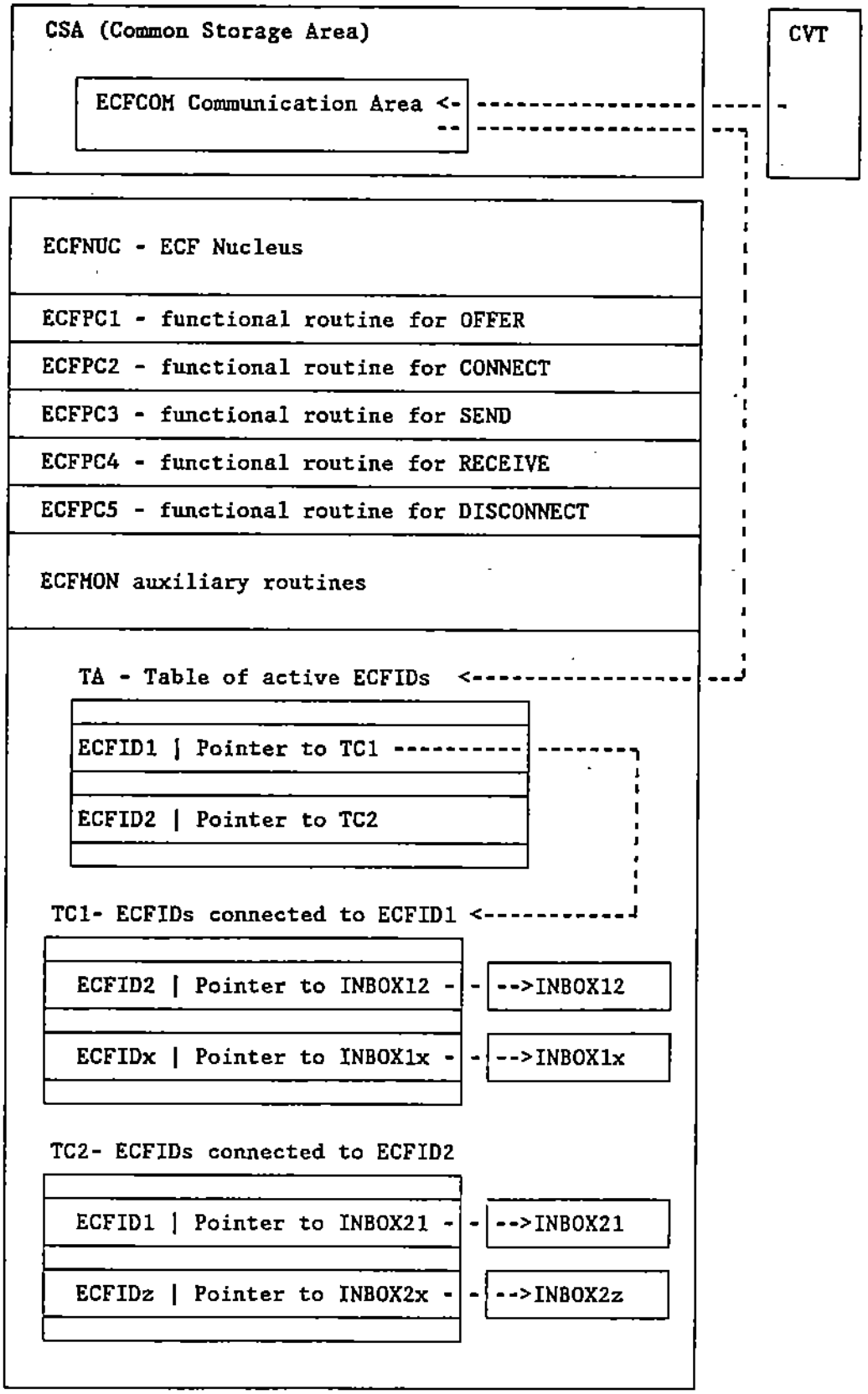

Figure 1. ECFMON structure and ECFAS map.