3. Diseño del controlador de emulación CAPI 2.0.

Problemas de compatibilidad entre la centralita y los controladores CAPI utilizados han condicionado fuertemente el desarrollo del proyecto. Tanto ha sido así que la mayor parte de los esfuerzos de diseño se ha dedicado a solventar este problema, relegando el diseño del programa servidor a un segundo plano.

A continuación se describe este problema y la solución adoptada. Debe quedar claro que no se está siguiendo un orden cronológico del desarrollo del proyecto, puesto que la incompatibilidad en cuestión se detectó durante las pruebas de un prototipo de programa servidor, y en ese momento la mayor parte del diseño -obviando la solución a este problema- estaba ya realizada. Utilizaremos esta descripción anacrónica para seguir en cierto modo el método de diseño *down-top* o *de abajo a arriba*, que personalmente considero más lógico para describir el desarrollo del proyecto.

3.1. Descripción del problema.

El sistema operativo Linux incluye controladores compatibles con el estándar CAPI 2.0 para tarjetas AVM activas. El calificativo activa significa aquí que la tarjeta incluye un *firmware* que se ocupa de la mayor parte de las tareas de la comunicación, lo que libera al procesador del equipo de las mismas.

La tarjeta AVM A1 utilizada no es activa, sino una tarjeta RDSI de tipo pasivo, por lo tanto los controladores incluidos con el núcleo de Linux no sirven. Para poder utilizar esta tarjeta, AVM proporciona un controlador CAPI 2.0 específico para el modelo A1 que puede obtenerse de forma gratuita del servidor `ftp` de AVM. En principio se utilizó dicho controlador con la versión del núcleo de Linux incluida en la distribución SuSE Linux 7.2.

CAPI 2.0 ofrece al programador la gestión del acceso RDSI mediante colas de mensajes, y esto incluye establecimiento y cierre de conexiones, transferencia de datos y acceso a los servicios suplementarios de la red. CAPI 2.0 proporciona facilidades para trabajar con todos los servicios ofrecidos por la red RDSI, tanto vocales como de datos digitales, fax, videoconferencia, etc.

Para el diseño del programa servidor `operait` se precisó de los servicios para establecimiento y cierre de conexión, transferencia de datos y los servicios suplementarios de detección pulsación de teclas, retención de llamada y transferencia explícita de llamada.

La gestión del estado de la conexión y la transferencia de datos tienen el objetivo claro de permitir aceptar llamadas, terminar llamadas y enviar datos vocales. No es necesario recibir los datos vocales del usuario puesto que no van a utilizarse: toda la comunicación con el servidor se realiza mediante la pulsación de teclas del teléfono, y para ello se utiliza el servicio suplementario CAPI de detección de pulsación de teclas. Finalmente, los servicios de retención de llamada y transferencia explícita de llamada se utilizan para desviar la llamada al número correspondiente, una vez el usuario ha seleccionado por teclado con quién quiere comunicarse. Son necesarios ambos servicios porque la norma Q.952.7, que describe el funcionamiento del servicio de transferencia explícita de llamada, exige que la llamada entrante a transferir se encuentre retenida antes de comenzar la transferencia.

El problema de incompatibilidad surgió al intentar utilizar el servicio de transferencia explícita en el bus RDSI de la centralita: la centralita Neris 64S consta de varios buses RDSI, pero no utiliza la norma Q.932 para ofrecer los servicios suplementarios RDSI. En su lugar, utiliza otro protocolo para ofrecer servicios similares, pero en la documentación disponible sobre la centralita no se indica qué protocolo es el utilizado. Por lo tanto el controlador CAPI 2.0 que ofrece AVM no permite la transferencia explícita de llamadas en el bus RDSI de la centralita Neris 64S.

3.2. El controlador CAPI 2.0 de AVM.

Para conseguir realizar la transferencia, se utilizó el terminal Crystal, que permite transferir llamadas en esos buses, conectado al analizador de redes HP Internet Advisor. Realizando varias pruebas y monitorizando los comportamientos de equipo terminal y terminador de red, se consiguió deducir el protocolo utilizado para transferir una llamada.

Una vez conocido el protocolo a utilizar, la solución al problema consistía en modificar el controlador CAPI para que su comportamiento coincidiese con el esperado por la centralita, tanto a nivel de formato de mensajes como de máquina de estados de la conexión. Hay que señalar que CAPI 2.0 no permite crear un mensaje arbitrario para enviarlo por la red, como por ejemplo ocurre con los *raw sockets* en la interfaz con los protocolos de acceso a Internet y red local. Si hubiera sido así, no habría existido la necesidad de modificar el controlador.

Por lo tanto, inicialmente todos los esfuerzos se dirigieron a realizar un proceso de ingeniería inversa sobre el código fuente del controlador para las tarjetas A1 proporcionado por AVM. Ésta no fue tarea sencilla puesto que, a pesar de distribuirse de forma gratuita, el código fuente carece totalmente de comentarios y está bastante mal estructurado.

Finalmente el controlador demostró ser imposible de modificar debido a que la parte del mismo que construye los mensajes a enviar por la red se distribuye como un archivo objeto precompilado y sin ficheros fuente. Precisamente las rutinas de construcción de mensajes Q.931 deben ser modificadas para poder añadir los mensajes comprensibles por la centralita. Por lo tanto esta posibilidad hubo de ser descartada.

3.3. Los controladores isdn4linux.

La solución definitiva al problema de crear un controlador adecuado para ejecutar transferencia de llamadas fue posible gracias a los controladores isdn4linux incluidos con el núcleo de Linux.

Estos controladores son bastante antiguos y han permanecido sin cambios durante varios años. Ofrecen servicios de acceso a la red RDSI para información vocal o para transferencia de datos digitales, y el uso principal que les dan los usuarios de Linux es para el acceso a Internet a través de conexiones PPP sobre RDSI con un Proveedor de Servicios de Internet (ISP).

Los controladores isdn4linux están implementados en dos partes: un controlador de nivel de enlace común a todas las tarjetas que gestiona los dispositivos `/dev/isdn*` y uno o varios controladores de bajo nivel específicos para cada tarjeta RDSI que mantienen el estado de las conexiones y la línea. En el caso de la AVM A1, el controlador de bajo nivel utilizado se llama `hisax.o` debido a que este controlador sirve para todas las tarjetas pasivas que utilicen el circuito integrado HiSax de Siemens.

Lo interesante de dichos controladores es que el interfaz de comunicación entre el controlador de nivel de enlace y el de bajo nivel está ampliamente documentado. Esto es así para que cualquier persona pueda crear un controlador de bajo nivel para una nueva tarjeta RDSI y éste pueda utilizarse con el controlador de nivel de enlace de isdn4linux.

La solución al problema consistía entonces en diseñar un controlador de nivel de enlace que reemplazase al original, manteniendo el interfaz con el controlador de bajo nivel. Este nuevo controlador tendría la función de realizar una *adaptación de capas*: en la interfaz con el usuario debería comportarse como la interfaz CAPI 2.0 y en la interfaz con el controlador de bajo nivel, como el controlador de nivel de enlace de isdn4linux. Además,

había que añadir al controlador de bajo nivel los mecanismos necesarios para realizar una transferencia de llamada usando el protocolo requerido por la centralita.

Aunque todo estas modificaciones no iban a resultar sencillas, al menos era posible realizarlas, gracias a la documentación incluida con la distribución del núcleo de Linux.

3.4. Proceso de transferencia de llamada.

La ilustración 2 muestra los tres primeros pasos de una transferencia de llamada.

- Inicialmente (1) existe una llamada establecida entre la aplicación OperAIT y un usuario externo, a través de la centralita Neris 64S.
- Cuando se desea realizar una transferencia de llamada, la aplicación OperAIT envía un mensaje de tipo `SETUP` por el mismo canal B a la centralita. Este mensaje incluye una facilidad de tipo `NETWORK SPECIFIC FACILITY` (2). Esto indica a la red que la llamada activa debe ser puesta en estado de espera (*hold*).
- El otro extremo aceptará la llamada y enviará el mensaje `ALERTING` a la centralita, que a su vez lo enviará a OperAIT (3).

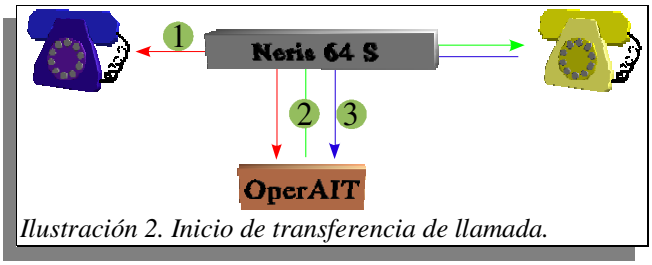

Ilustración 2. Inicio de transferencia de llamada.

En este instante existen dos llamadas, una establecida y otra en fase de establecimiento. La primera de ellas, entre OperAIT y el usuario inicial de la aplicación, actualmente se encuentra en estado de espera. La segunda aún no ha completado su establecimiento, puesto que el extremo remoto no ha contestado a OperAIT.

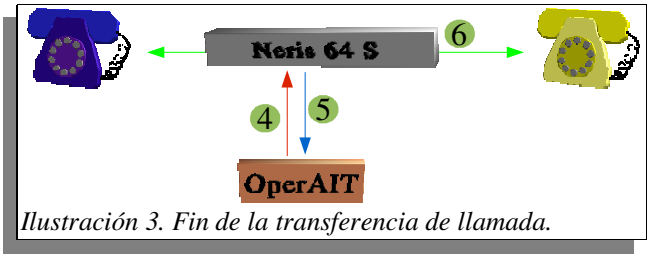

Ilustración 3. Fin de la transferencia de llamada.

La ilustración 3 muestra el fin de la transferencia de llamada: la conexión aún no establecida y la llamada en espera formarán una sola conexión por establecer y la aplicación OperAIT liberará las conexiones con sus dos extremos. De este modo los dos usuarios, el usuario llamante original y el usuario al que se envía la segunda llamada, se comunicarán como si el primero hubiera realizado la llamada al segundo.

Para ello, es necesario que OperAIT, una vez reciba el mensaje `ALERTING`, envíe a la centralita un mensaje de tipo `DISCONNECT` (4). La centralita finalizará ambas conexiones (5) y a partir de este momento las dos llamadas iniciales formarán una sola (6), quedando OperAIT listo para atender nuevas llamadas.

Este caso es el ideal, pero pueden darse las siguientes desviaciones de dicho comportamiento:

- Se recibe un mensaje `DISCONNECT` en el paso 3, en lugar del `ALERTING` esperado. Esto puede ocurrir porque se esté intentando transferir la llamada a un número inexistente o por problemas en la red.
- Se recibe un mensaje `SETUP ACKNOWLEDGE` en el paso 3, en lugar del `ALERTING` esperado. Esto significa que la red continúa esperando cifras del número de teléfono al que transferir la llamada. Puesto que el usuario debe pasar como parámetro la totalidad de las cifras del número, esto sólo puede significar que se está intentando transferir la llamada a un número inexistente[1].

En ambos casos será imposible establecer la segunda llamada, y el comportamiento del sistema será el mismo: terminará las dos comunicaciones. Lo más deseable sería que OperAIT reprodujese un mensaje indicando que hubo un error interno al usuario inicial del sistema, pero no ha sido posible conseguir restaurar una llamada que haya sido previamente puesta en espera, por lo que la única solución posible es desconectarla.

3.5. Modificaciones al controlador de bajo nivel.

3.5.1. Comunicación entre ambos controladores.

Antes de comenzar a comentar las distintas modificaciones que hay que realizar al controlador de bajo nivel, se ofrece una descripción somera del método utilizado para comunicar cada una de las partes del subsistema isdn4linux.

En un mismo sistema pueden existir varios controladores de bajo nivel si hay distintos modelos de tarjetas RDSI trabajando simultáneamente. Para permitir la comunicación entre el controlador de nivel de enlace y cada uno de los posibles controladores de bajo nivel, estos últimos deben *registrarse* antes de estar plenamente operativos.

El proceso de registro de un controlador de bajo nivel requiere que el controlador de nivel de enlace forme parte del núcleo, lo que puede conseguirse bien incluyéndolo como parte estática del núcleo o compilándolo como un módulo e insertándolo con los comandos

[1] En esta caso, será un número con insuficientes cifras como para identificar a un abonado o señalar un error por número inexistente. Por ello la centralita queda a la espera de subsecuentes cifras.

`insmod` o `modprobe`. El controlador de nivel de enlace exporta la siguiente función que sirve para el registro:

```
int register_isdn (isdn_if *hldrv);
```

El controlador de bajo nivel, al iniciarse, debe realizar una llamada a la función anterior. El parámetro `hldrv` debe ser un puntero a una estructura `isdn_if`, declarada en el archivo `include/linux/isdnif.h` distribuido con el código fuente del núcleo de Linux. Ciertos campos de esta estructura deben ser rellenados por el controlador de bajo nivel, y el resto debe ser rellenado por el de nivel de enlace; `hldrv` es un argumento de entrada/salida. Los campos importantes para la comunicación entre ambos controladores son:

```
void (*rcvcallb_skb)(int, int, struct sk_buff *);
int (*statcallb)(isdn_ctrl*);
int (*command)(isdn_ctrl*);
```

Los dos primeros, `rcvcallb_skb` y `statcallb` son parámetros de retorno proporcionados por el controlador de nivel de enlace y son punteros funciones que el controlador de bajo nivel debe utilizar para indicar recepción de nuevos datos y cambio de estado en la conexión, respectivamente. El último parámetro, `command`, es un puntero a función proporcionado por el controlador de bajo nivel que el controlador de nivel de enlace debe utilizar para enviarle los comandos que gestionan las conexiones.

3.5.2. Nueva funcionalidad.

Resumiendo, el controlador de nivel de enlace que hay que diseñar debe incluir dos funciones con los prototipos anteriormente vistos para recibir indicaciones de datos y cambio de estado del controlador de bajo nivel. Asimismo, los comandos que controlan el estado de la conexión deben ser enviados al controlador de bajo nivel mediante el uso del puntero a función `command`, una vez el mismo se haya registrado.

Precisamente los cambios a realizar en el controlador de bajo nivel van dirigidos a proporcionar un nuevo comando para realizar transferencia de llamada y a enviar los mensajes adecuados y ampliar la máquina de estados de la conexión para añadir esta nueva funcionalidad. Los cambios realizados son:

- Añadido del comando `CC_ENHANCED` a la lista de comandos comprensibles para el controlador de bajo nivel. Dicho comando se utiliza para realizar la transferencia de llamada.
- Supresión del envío de mensajes Q.931 de tipo `STATUS` a la red. Esto debe hacerse porque el controlador original no entiende los mensajes que utiliza la centralita para

realizar la transferencia, con lo que contestaría a éstos con un mensaje de tipo `STATUS` que finalizaría prematuramente la transferencia de llamada.

- Ampliación de la máquina de estados de conexión mediante la definición del evento `EV_ENHANCED` y nuevas funciones y variables para mantener el estado de la comunicación durante el intento de transferencia.
- Definición de funciones para crear los mensajes de transferencia de llamada y procesar las respuestas de la centralita.

Los archivos del núcleo de Linux que han de ser modificados son `callc.c`, `hisax.h`, `isdnif.h` y `l3dss1.c`. Cambiar estos archivos no supone ninguna incompatibilidad con los controladores originales salvo el hecho de que un mensaje no reconocido por el nuevo controlador no causará el envío de un mensaje `STATUS` como ocurría con el antiguo. Nuevas versiones de estos controladores podrían ser incompatibles pero, puesto que hace varios años que no se modifican, es un caso bastante improbable por el momento.

Tras estas modificaciones existirá el comando `CC_ENHANCED`, mediante el cual el controlador de nivel de enlace indicará al de bajo nivel que se desea realizar una transferencia explícita de llamada. Por su parte, el controlador de bajo nivel señalará la finalización de la transferencia, indicando si se realizó correctamente o si hubo algún fallo. En ambos casos, la llamada a transferir terminará. Aunque con el teléfono Crystal es posible recuperar una llamada tras un intento fallido de transferencia, al intentarlo con la tarjeta RDSI no se ha podido emular dicho comportamiento de forma efectiva. Por lo tanto un intento de transferencia de llamada terminará la llamada activa.

3.5.3. Implementación de las máquinas de estado.

Es importante comprender el mecanismo para gestionar los estados del sistema que utiliza el controlador de bajo nivel, puesto que ha sido necesario modificarlo para añadir la nueva funcionalidad.

La máquina de estados de la comunicación se implementa como una tabla de estructuras de tipo `FsmNode`. Este tipo de datos se define en el archivo `hisax.c`, y consiste básicamente en dos números enteros que indican el estado y el evento que genera una transición en dicho estado, junto con un puntero a función que es la que gestiona el cambio de estado.

Para añadir una nueva transición entre estados a una máquina de estados de este tipo será necesario indicar un número que identifique el estado anterior a la transición, otro entero

que indique el evento que causa la misma, y definir la función que realice los cambios necesarios -como, por ejemplo, modificar el estado actual-.

La definición del tipo `FsmNode` es:

```
struct FsmNode {
        int state, event;
        void (*routine) (struct FsmInst *, int, void *);
};
```

A cada posible estado del sistema se le asocia un valor de tipo entero, al igual que ocurre con cada uno de los eventos significativos. Por sencillez y claridad, se definen dos enumeraciones, una para los estados y otra para los eventos. Como ejemplo, la enumeración asociada a los estados en el archivo `callc.c` es:

```
enum {
        ST_NULL,                /*  0 inactive */
        ST_OUT_DIAL,            /*  1 outgoing, SETUP send; awaiting
                                      confirm */
        ST_IN_WAIT_LL,          /*  2 incoming call received; wait
                                    for LL confirm */
        ST_IN_ALERT_SENT,       /*  3 incoming call received; ALERT
                                    send*/
        ST_IN_WAIT_CONN_ACK,    /*  4 incoming CONNECT send; awaiting
                                   CONN_ACK */
        ST_WAIT_BCONN,          /*  5 CONNECT/CONN_ACK received,
                                    awaiting b-channel prot. est. */
        ST_ACTIVE,              /*  6 active,b channel prot.
                                    established*/
        ST_WAIT_BRELEASE,       /*  7 call clear. (initiator),
                                    awaiting b channel prot. rel. */
        ST_WAIT_BREL_DISC,      /*  8 call clear. (receiver),
                                    DISCONNECT req. received */
        ST_WAIT_DCOMMAND,       /*  9 call clear. (receiver),
                                    awaiting DCHANNEL message */
        ST_WAIT_DRELEASE,       /* 10 DISCONNECT sent, awaiting
                                   RELEASE */
        ST_WAIT_D_REL_CNF,      /* 11 RELEASE sent, awaiting RELEASE
                                   confirm */
        ST_IN_PROCEED_SEND,     /* 12 incoming call, proceeding
                                   send */
};
```

Una vez se haya escrito la función que gestiona un cambio de estado a partir de un estado inicial debido a un evento determinado, se introducirá una entrada en la tabla `fnlist`. Como ejemplo, el principio de esta tabla en el archivo `callc.c` es:

```
static struct FsmNode fnlist[] __initdata =
{
        {ST_NULL,       EV_DIAL,                lli_prep_dialout},
        {ST_NULL,       EV_RESUME,              lli_resume},
        {ST_NULL,       EV_SETUP_IND,           lli_deliver_call},
    /* Continúa... */
}
```

En el archivo `hisax.h` se define la función `FsmEvent` que, tomando como parámetros la estructura que mantiene el estado actual y el evento recibido, recorre la tabla anterior para comprobar si dicho evento dispara algún cambio de estado y en tal caso ejecuta la función que gestiona el mismo.

Con todos estos componentes, realizar una máquina de estados es bastante más sencillo que utilizar construcciones `switch`/`case`.

3.5.4. Modificaciones al archivo callc.c.

El archivo `callc.c` proporciona la máquina de estados interna del controlador de bajo nivel HiSax. La comunicación entre el controlador de bajo nivel y el controlador de nivel de enlace se gestiona mediante el código de este archivo.

Este archivo incluye una función, llamada `HiSax_command`, que será la que utilice el controlador de nivel de enlace para enviar comandos al controlador de bajo nivel. Todo esto se realizará a través del puntero a esta función que el controlador de bajo nivel proporciona al controlador de nivel de enlace cuando realiza el registro.

Además de la función `HiSax_command`, es importante la tabla de transiciones entre estados, que utiliza el sistema comentado en el apartado anterior. Cada vez que se recibe un evento, bien porque se reciba un comando proveniente del controlador de nivel de enlace bien porque se reciba desde la parte del controlador de bajo nivel que trata directamente con la tarjeta RDSI, se realiza una llamada a la función `FsmEvent` para indicar la recepción del mismo.

Los cambios realizados al archivo `callc.c` van encaminados a conseguir una nueva transición de estados desde el estado `ST_ACTIVE` (conexión activa). Para ello se añade a la tabla de transición de estados la siguiente entrada:

```
{ST_ACTIVE,             EV_ENHANCED,            lli_enhanced}
```

También se definen las funciones `lli_enhanced` y `lli_enhanced_conf`. La primera de ellas gestiona el evento de petición de transferencia de llamada recibido del controlador de nivel de enlace. La segunda, recibe el valor de retorno del intento de transferencia de llamada

de las subrutinas que controlan la tarjeta RDSI y envía el mismo al controlador de nivel de enlace.

Se declara a su vez dos variables para almacenar el número de teléfono del interlocutor para cada canal B mientras se intenta la transferencia de llamada. Esto es necesario puesto que el número al que se desea transferir sobrescribe al número con el que se mantiene la comunicación, y no se desea perder esta información cuando se intente una transferencia.

Por lo tanto, el proceso de intento de transferencia de llamada es visto como sigue por el código de este archivo:

- El controlador de nivel de enlace envía el comando `ISDN_CMD_ENHANCED` al controlador de bajo nivel, a través del puntero a función que apunta a `HiSax_command`.
- La función `HiSax_command` almacena el número del interlocutor actual en la variable destinada a ello, sobrescribe el número del interlocutor actual con el número al que se desea transferir la llamada e invoca a la función `FsmEvent` con el evento `EV_ENHANCED` como parámetro.
- La función `FsmEvent` recorre la tabla de transición de estados y comprueba que en el estado actual (`ST_ACTIVE`) el evento `EV_ENHANCED` produce una transición gestionada por la función `lli_enhanced`, por lo que invoca a la misma.
- La función `lli_enhanced` ordena al módulo que envía los mensajes por la red que ejecute el comando `CC_ENHANCED` con subcomando `REQUEST`. Esto se consigue mediante una llamada a la función `lli.l4l3`. A partir de aquí entra en acción código que pertenece al archivo `l3dss1.c`. Cuando éste termina, termina la función `lli_enhanced`, termina la función `FsmEvent` y termina la función `HiSax_command`. El intento de transferencia ha comenzado.
- Más adelante, cuando la red haya devuelto un valor de retorno de la operación, la función `dchan_l3l4` será invocada por el módulo que controla la tarjeta de red, que indicará el evento `CC_ENHACED` con sub-evento `CONFIRMATION`. La función `dchan_l3l4` comprobará estos valores y realizará una llamada a la función `lli_enhanced_conf`.
- La función `lli_enhanced_conf` toma el valor de retorno recibido e indica el estado `ISDN_STAT_ENHANCED` al controlador de nivel de enlace a través del puntero a función `statcallb`. Este puntero a función se obtuvo en el proceso de registro del

controlador de bajo nivel. A partir de aquí, el controlador de nivel de enlace deberá actualizar su estado utilizando este valor de retorno.

Puede verse que el proceso de transferencia de una llamada no es sencillo, puesto que involucra tanto al controlador de nivel de enlace como a distintos módulos del controlador de bajo nivel y cada una de estas comunicaciones tiene un método propio de iniciación y paso de parámetros.

3.5.5. Modificaciones al archivo l3dss1.c.

Las modificaciones que se han realizado al archivo `l3dss1.c` son más sencillas que en el caso anterior. Lo único que hay que conseguir es poder crear los mensajes compatibles con la centralita Neris 64S para transferencia explícita de llamada, y procesar las respuestas de la misma de forma correcta.

En este caso todo es más sencillo porque la comunicación entre este módulo y el resto del controlador de bajo nivel se realiza mediante las funciones `dss1up` y `dss1down`. La primera de ella recibe indicaciones del código que controla directamente la tarjeta. La segunda comunica el módulo implementado en el archivo `l3dss1.c` con el código incluido en el archivo `callc.c`.

Es necesario modificar la función `dss1down` para que, cuando el código incluido en el archivo `callc.c` envíe el comando `CC_ENHANCED|REQUEST`, se invoque a la función `l3dss1_enhanced_req`. Esta función, que también ha sido añadida al código original, crea el mensaje Q.931 a enviar a la centralita para realizar la transferencia de llamada y lo pasa a las funciones que lo envían a la tarjeta RDSI. Además de ello, utiliza variables que se han añadido para mantener el estado, indicando que se ha comenzado un intento de transferencia de llamada y almacenando el identificador de la llamada a transferir.

Las modificaciones realizadas a `dss1up` se basan en el identificador de llamada (*Call Reference*) de RDSI. Se ha comentado que cuando se recibe una petición de transferencia de llamada se almacena el número de referencia de la llamada a transferir. A partir de entonces, cada vez que se ejecute la función `dss1up`, se comprueba si el mensaje tiene un valor de referencia de llamada coincidente con el almacenado. Si es así, se pasa el control a la función añadida `l3dss1_handle_ect`, que gestiona la transferencia de llamada, evitando que el sistema procese dicho mensaje. Esta función comprueba el tipo de mensaje recibido y actúa en consecuencia.

Un intento de transferencia de llamada causaría la siguiente cadena de ejecución:

- El código del archivo `callc.c` envía el comando `CC_ENHANCED|REQUEST` como parámetro para la función `dss1down`. Esta función comprueba el código de comando e invoca a la función `l3dss1_enhanced_req`.
- La función comprueba que no se estaba intentando transferir una llamada por el canal B indicado y crea el mensaje Q.931 `SETUP` incluyendo una facilidad de tipo *Network Specific Facility* para indicar a la centralita que se desea realizar una transferencia explícita de llamada. Asimismo, almacena el número de referencia de la llamada a transferir y modifica las variables añadidas que almacenan el estado de la transferencia para dicho canal B. El mensaje se crea en una estructura `sk_buff` y se utiliza la función `l3_msg`, que se ocupará de enviarlo a la tarjeta RDSI.
- Cuando la red envíe un mensaje, se comprobará si la referencia de llamada asociada coincide con el valor almacenado. Si es así, la función `l3dss1_handle_ect` se ejecutará.
- Según el tipo de mensaje recibido de la red, la acciones que se llevarán a cabo pueden incluir enviar mensajes a la red, señalizar cambios al nivel superior o modificar variables de estado. El comportamiento para cada mensaje recibido es:
- `RELEASE`: se creará el mensaje `RELEASE_COMPLETE` correspondiente y se comprueba si hubo algún error en el proceso de transferencia. Se envía el mensaje `CC_ENHANCED | CONFIRM` al nivel superior, indicando el código de error si lo hubo.
- `ALERTING`: este mensaje indica que la llamada saliente se ha establecido y está a la espera de que el otro extremo la conteste, por lo que se enviará el mensaje `DISCONNECT` para que la red desconecte al equipo de la red y transfiera la llamada.
- `RELEASE_COMPLETE`: la transferencia habrá terminado en este momento, ya sea con error o exitosamente, por lo que se modifican las variables que indican transferencia en el canal B correspondiente y almacenan la referencia de llamada a transferir.
- `DISCONNECT`: este mensaje indicará un error, por lo que se halla la causa de desconexión y se envía el mensaje `RELEASE`.
- `SETUP_ACKNOWLEDGE`: este mensaje indica que la red está a la espera de más cifras del número al que transferir, y puesto que no se conocen más cifras de este número, el usuario habrá intentado transferir la llamada a un número de insuficientes cifras. Se creará entonces el mensaje `RELEASE` y se enviará al nivel superior el mensaje `CC_ENHANCED | CONFIRM`, indicando la causa de desconexión `INVALID_NUMBER_CAUSE`.

- Según el estado del intento de transferencia de llamada, se seguirán recibiendo mensajes que gestionará la función `l3dss1_handle_ect`, terminando este proceso cuando se envíe el mensaje `RELEASE_COMPLETE`.

3.6. El nuevo controlador de nivel de enlace.

Una vez el controlador de bajo nivel ofrece el servicio de transferencia de llamada, gracias a las modificaciones anteriores, hemos de crear un controlador que emule la interfaz CAPI 2.0. El objetivo de dicho controlador es traducir los mensajes CAPI a comandos para el controlador de bajo nivel y las indicaciones de éste a mensajes CAPI para el usuario.

El interfaz CAPI 2.0 está basado en un número mínimo de funciones para trabajar con colas de mensajes. Existe una cola de mensajes entrante a la que todos programas registrados envían sus mensajes a CAPI 2.0, y una cola de mensajes por aplicación registrada a la que CAPI 2.0 envía todos los mensajes para dicha aplicación.

La emulación CAPI 2.0 diseñada nos proporciona las funciones requeridas y se comporta de forma similar al controlador CAPI 2.0 estándar, pero no implementa la mayoría de la funcionalidad de éste. Esto es así porque es un controlador altamente experimental y cuya única función es solventar la incompatibilidad de protocolo entre centralita y controlador estándar. No tiene intención de convertirse en un controlador portable y utilizable en otros sistemas porque diseñarlo para ello sería demasiado complicado y saldría del objetivo del proyecto. Todo ha sido diseñado de la forma más sencilla posible, y sólo se ha escrito el código que era estrictamente necesario para el funcionamiento de la aplicación OperAIT.

La primera diferencia con el controlador CAPI 2.0 estándar estriba en que sólo se permite que exista una aplicación utilizando la tarjeta RDSI. Además, sólo se ha emulado la máquina de estados CAPI 2.0 para llamadas entrantes, puesto que OperAIT no genera llamadas salientes -salvo para la transferencia de llamadas, pero eso no pertenece a la máquina de estados CAPI 2.0-. El único tipo de llamadas entrantes que se permite es de datos vocales: no es posible enviar fax, datos digitales, vídeo, etc. En cambio, sí se implementa la facilidad CAPI 2.0 para detección de tonos de teclado -DTMF-, puesto que es esencial para que el usuario pueda seleccionar las opciones del menú.

Por otro lado, la emulación de CAPI 2.0 no sólo consiste en un controlador de nivel de enlace corriendo en espacio de núcleo, también existe una parte que corre en espacio de memoria de usuario. Esta parte es una sencilla biblioteca que se enlaza estáticamente a la aplicación final y es la que proporciona las funciones `CAPI_REGISTER`, `CAPI_PUT_MESSAGE`, etc.

Las funciones realizadas por el controlador de nivel de enlace son las siguientes:

- Mantener la máquina de estados de todo el sistema.
- Realizar detección de tonos multifrecuencia -DTMF- sobre los datos recibidos y enviar a la aplicación el mensaje correspondiente si se detecta alguna pulsación de tecla.
- Crear y mantener la cola de mensajes para la aplicación.
- Descodificar los mensajes que envía la aplicación y transformarlos en comandos para el controlador de bajo nivel.
- Transformar las indicaciones de estado del controlador de bajo nivel y transformarlos en mensajes CAPI 2.0 para la aplicación.
- Proporcionar los dispositivos en el directorio `/dev` asociados a CAPI 2.0, incluyendo todas las operaciones posibles (`read`, `write`, etc.).
- Definir la función `register_isdn` para gestionar el registro del controlador de bajo nivel, almacenando los campos necesarios para la comunicación con el mismo.

Como puede verse, las funciones no son pocas, y conviene tratarlas por separado. Por ello existen varios archivos fuente cada uno de los cuales realiza una función específica o varias de ellas que estén relacionadas.

3.6.1. El archivo inicio.c.

El controlador de emulación CAPI está compilado como un módulo de núcleo -*Loadable Kernel Module*- cuyo nombre es `rdsi.o`. Este módulo debe ser cargado antes de insertar el módulo correspondiente al controlador de bajo nivel modificado, `hisax.o`. El archivo `inicio.c` incluye todo el código que se ocupa de la carga y descarga del módulo mediante las funciones `init_module` y `cleanup_module`. Además define la función `register_isdn`, mediante la cual se registrará el controlador de bajo nivel.

El código que gestiona el registro del controlador de bajo nivel no comprueba si existía anteriormente otro controlador registrado y está diseñado exclusivamente para tratar con un controlador de bajo nivel, por lo que no debe intentarse insertar dos de estos controladores si se utiliza el módulo `rdsi.o`.

3.6.2. El archivo callback.c.

Este archivo contiene el código correspondiente a las funciones de *callback*, es decir, las funciones mediante las que el controlador de bajo nivel notifica los cambios de estado y recepción de datos.

Estas funciones, según el identificador de estado recibido y si corresponde, realizarán llamadas a las funciones `Handle_ISDNBdata` y `Handle_ISDN_msg`, que se definen en el archivo `msghandle.c` y que controlan la máquina de estados del sistema.

3.6.3. El archivo msghandle.c.

Toda la gestión de la máquina de estados del sistema se realiza por el código guardado en este archivo. Las rutinas del mismo procesan las indicaciones provenientes del controlador de bajo nivel y de la aplicación y actúan modificando el estado, creando mensajes CAPI para la aplicación y enviando comandos al controlador de bajo nivel. También se declaran aquí las variables correspondientes al estado del sistema.

La máquina de estados está implementada como una tabla de estructuras de tres campos, de forma similar a las máquinas de estado del controlador de bajo nivel de RDSI. Los dos primeros indican el estado y el evento que provoca un cambio si el sistema se encuentra en ese estado. El último de ellos corresponde a un puntero a función que apunta a la rutina que gestiona el cambio de estado. De esta forma, para añadir una nueva transición de estados basta con añadir una entrada a esta tabla y definir la función que realice las acciones correspondientes a dicha transición.

Los mensajes CAPI dirigidos a la aplicación se crean aquí mediante el uso de `sk_buffers`, las estructuras de datos estándar para almacenar mensajes del subsistema de red del núcleo de Linux. Una vez el mensaje ha sido almacenado en un `sk_buffer`, este último se inserta en una cola de mensajes, a la espera de que la aplicación lo retire mediante la llamada a la función `CAPI_GET_MESSAGE`. Esta cola de mensajes no está declarada en este archivo, sino en ella archivo `capidev.c`. `msghandle.c` es, con mucho, el archivo más complejo y extenso de los que componen el controlador de nivel de enlace.

Las funciones más importantes de este módulo son las que gestionan la recepción de eventos y/o transiciones en la máquina de estados:

- `Handle_CAPI_msg` gestiona los eventos recibidos por la interfaz con el usuario[2] y los cambios de transición en el estado debidos a ellos. Será la función que tiene asociada cada cambio de estado la que cree los mensajes CAPI 2.0 para el usuario, según el estado al que se llegue.
- `Handle_ISDN_msg` gestiona los eventos recibidos del controlador de bajo nivel y de modificar el estado del sistema en consecuencia. Estos eventos son todos aquellos

[2] Estos eventos los genera el usuario al utilizar la función `CAPI_PUT_MESSAGE`, lo que se traduce finalmente en realizar la llamada al sistema `WRITE` sobre el dispositivo `/dev/capi20`.

que se describe en la documentación de la interfaz de comunicación de los controladores isdn4linux.

- `Handle_ISDNBdata` gestiona la recepción de datos desde la red y los transforma en mensajes CAPI 2.0 de tipo `CAPI_DATA_B3 INDICATION` para el usuario.

3.6.4. El archivo command.c.

Este sencillo archivo contiene el código de varias funciones bastante simples, una por cada comando que el controlador de nivel de enlace puede enviar al controlador de bajo nivel. Éstas serán invocadas desde las funciones que gestionan las transiciones de estado, en general como respuesta a mensajes CAPI procedentes de la aplicación.

3.6.5. El archivo capidev.c.

La emulación CAPI se basa, como todo el acceso a *hardware* en Linux, en la creación de dispositivos. El controlador de nivel de enlace gestiona el dispositivo de caracteres `/dev/capi20`, y todo el código que realiza este proceso se encuentra en `capidev.c`.

Este código define las funciones que atenderán las llamadas al sistema `OPEN`, `CLOSE`, `READ`, `WRITE`, `POLL`, etc. También la iniciación y la descarga del dispositivo se realizan aquí. La forma de realizar esta operación es común a todos los dispositivos de caracteres de Linux, y consiste en utilizar la función `register_chrdev`, a la que se le pasan como parámetros el número mayor (*major number*) y el nombre del dispositivo de caracteres y un puntero a una estructura `file_operations`. Esta estructura se define en el archivos `include/linux/fs.h` dentro del árbol de código del núcleo de Linux y sus campos más importantes son los punteros a función que se declaran:

```
struct file_operations {
      struct module *owner;
      loff_t (*llseek) (struct file *, loff_t, int);
      ssize_t (*read) (struct file *, char *, size_t, loff_t *);
      ssize_t (*write) (struct file *, const char *, size_t,
                loff_t *);
      int (*readdir) (struct file *, void *, filldir_t);
      unsigned int (*poll) (struct file *,
                     struct poll_table_struct *);
      int (*ioctl) (struct inode *, struct file *, unsigned int,
            unsigned long);
      int (*mmap) (struct file *, struct vm_area_struct *);
      int (*open) (struct inode *, struct file *);
      int (*flush) (struct file *);
      int (*release) (struct inode *, struct file *);
```

```
      int (*fsync) (struct file *, struct dentry *, int datasync);
      int (*fasync) (int, struct file *, int);
      int (*lock) (struct file *, int, struct file_lock *);
      ssize_t (*readv) (struct file *, const struct iovec *,
                unsigned long, loff_t *);
      ssize_t (*writev) (struct file *, const struct iovec *,
                unsigned long, loff_t *);
      ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
                loff_t *, int);
      unsigned long (*get_unmapped_area)(struct file *,
                      unsigned long, unsigned long, unsigned long,
                      unsigned long);
};
```

Mediante la creación de este dispositivo, las funciones de biblioteca `CAPI_REGISTER`, `CAPI_RELEASE`, `CAPI_PUT_MESSAGE`, `CAPI_GET_MESSAGE` y `CAPI_WAIT_FOR_MESSAGE` se convertirán en llamadas al sistema `OPEN`, `CLOSE`, `WRITE`, `READ` y `POLL` -respectivamente- sobre el dispositivo `/dev/capi20`. Cuando se realicen estas llamadas al sistema, el núcleo otorgará el control a las funciones que se hayan pasado como parámetro a `register_chrdev`.

Las funciones que define este archivo, la llamada al sistema asociada y su modo de operación se indican en la siguiente tabla.

Llamada al sistema	*Función*	*Acción realizada*
`OPEN`	`device_open`	Inicia las estructuras internas del estado de la emulación CAPI 2.0.
`CLOSE`	`device_release`	Libera las conexiones activas e inicia el estado de la emulación CAPI 2.0.
`WRITE`	`device_write`	Toma el mensaje escrito por el proceso de usuario y lo pasa como parámetro a `Handle_CAPI_msg`.
`READ`	`device_read`	Si hay mensajes en la cola para la aplicación, devuelve el primero de ellos al proceso de usuario.
`POLL`	`device_select`	Duerme el proceso de usuario hasta que la cola de mensajes deja de estar vacía.
`IOCTL`	`device_ioctl`	No implementada, sólo para depuración.

Tabla 1. Llamadas al sistema y funciones que las tratan.

Además de manejar las llamadas al sistema sobre el dispositivo, se declara la cola para almacenar los `sk_buffers` con los mensajes para la aplicación y se define y exporta la función utilizada para añadir un mensaje a la cola. No es necesario crear una cola de entrada para almacenar los mensajes de la aplicación puesto que éstos se procesan inmediatamente

después de que se realice la llamada al sistema `WRITE` sobre el dispositivo `/dev/capi`. Así pues la emulación CAPI difiere de la biblioteca CAPI 2.0 original en que no existe una cola para los mensajes de entrada, como ocurre con ésta.

3.6.6. El archivo dtmfdetect.c.

Para proporcionar el servicio de detección de tonos de teclado o tonos DTMF, se necesita realizar un análisis espectral de los datos de entrada. Este archivo proporciona las funciones necesarias para ello.

El sistema utilizado para detectar pulsación de teclas es calcular la Transformada Discreta de Fourier -DFT- de un bloque de datos vocales. El algoritmo utilizado comúnmente para ello, debido a su simplicidad, es el algoritmo de Goertzel. Este algoritmo se basa en varias relaciones matemáticas existentes entre una secuencia y su DFT, como veremos a continuación.

La transformada discreta de Fourier -DFT- de una secuencia x[n] de longitud N muestras se define como:

$$\varphi[k]=\sum_{n=0}^{N-1} x[n]\,\mathrm{e}^{-j\frac{2\pi kn}{N}}=\sum_{n=0}^{N-1} x[n]\,W_N^{kn} \qquad donde\ W_N^{kn}=\mathrm{e}^{-j\frac{2\pi kn}{N}}$$

Esto nos lleva a la siguiente relación:

$$W_N^{-kN}=1 \;\Rightarrow\; \varphi[k]=W_N^{-kN}\sum_{r=0}^{N-1} x[r]\,W_N^{kr}=\sum_{r=0}^{N-1} x[r]\,W_N^{-k(N-r)}$$

El sumatorio coincide con la muestra N del resultado de realizar la suma de convolución entre x[n] y un filtro de respuesta al impulso:

$$h_k[n]=W_N^{-kn}u[n]$$

Definiendo entonces la secuencia resultado de la convolución x[n]*h_k[n]:

$$y_k[n]=\sum_{r=\infty}^{\infty} x[r]\,W_N^{-k(n-r)}u[n-r]$$

Se cumple, según hemos visto:

$$\varphi[k]=y_k[n]_{n=N}=y_k[N]$$

En resumen, para conseguir hallar la muestra k de la DFT de la secuencia de entrada, basta con aplicar a la entrada x[n] el filtro con respuesta al impulso h_k[n] y tomar la muestra N de la salida.

Por lo tanto, si queremos hallar M muestras de la DFT, hay que aplicar a la secuencia de entrada M filtros distintos y tomar las muestras en posición N de cada una de las M salidas. Esto no es ni mucho menos óptimo. El algoritmo de Goertzel supone demasiada carga

computacional y no es tan avanzado como los algoritmos de transformada rápida de Fourier -FFT-, pero se utiliza mucho porque es sencillo de implementar, debido a lo siguiente:

La transformada Z de el filtro con respuesta $h_k[n]$ es:

$$H_k[z]=\frac{1}{1-W_n^{-k}z^{-1}}=\frac{1-W_N^k z^{-1}}{(1-W_n^k z^{-1})(1-W_n^{-k}z^{-1})}=\frac{1-W_N^k z^{-1}}{1-2z^{-1}\Re[W_N^k]+z^{-2}}$$

$$W_N=e^{-j\frac{2\pi}{N}} \Rightarrow \Re[W_N]=\cos(\frac{2\pi k}{N}) \Rightarrow H_k[z]=\frac{1-W_N^k z^{-1}}{1-2z^{-1}\cos(\frac{2\pi k}{N})+z^{-2}}$$

Relacionando $H_k[z]$ con las transformadas Z de entrada, salida y el *estado* del sistema w[n], tenemos:

$$H_k[z]=\frac{Y_k[z]}{X[z]}=\frac{Y_k[z]}{W[z]}\cdot\frac{W[z]}{X[z]}=1-W_N^k z^{-1}\cdot\frac{1}{1-2z^{-1}\cos(\frac{2\pi k}{N})+z^{-2}}$$

Y tomando las relaciones siguientes para X[z] y W[z]:

$$\frac{W[z]}{X[z]}=\frac{1}{1-2z^{-1}\cos(\frac{2\pi k}{N})+z^{-2}} \Rightarrow$$

$$W[z]=X[z]+2z^{-1}\cos(\frac{2\pi k}{N})W[z]-z^{-2}W[z] \Rightarrow$$

$$w[n]=x[n]+2\cos(\frac{2\pi k}{N})w[n-1]-w[n-2]$$

Y las siguientes para $Y_k[z]$ y W[z]:

$$\frac{Y_k[z]}{W[z]}=1-W_N^k z^{-1} \Rightarrow$$

$$Y_k[z]=W[z]-W_N^k z^{-1}W[z] \Rightarrow$$

$$y_k[n]=w[n]-W_N^k w[n-1]=w[n]-\cos(\frac{2\pi k}{N})w[n-1]+i\,sen(\frac{2\pi k}{N})w[n-1]$$

Ambas ecuaciones nos llevan al siguiente grafo de flujo de señal:

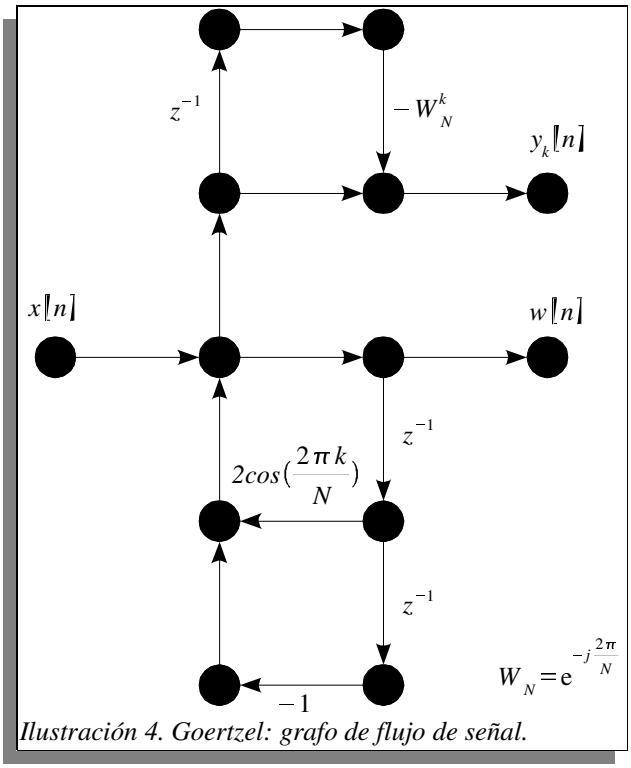

Ilustración 4. Goertzel: grafo de flujo de señal.

Los cálculos de la rama superior del grafo sólo hay que realizarlos para n=N, puesto que la única muestra de la salida $y_k[n]$ que nos interesa es ésa. Para las ramas inferiores sí hay que realizar los cálculos para todas las muestras, ya que mantienen el estado del sistema -existe realimentación-.

El código incluido en el archivo realiza una implementación en C del algoritmo anterior utilizando tipo de datos `short int` y desplazamientos de bits para emular aritmética en coma fija. Lo que en realidad se calcula es el módulo del valor complejo y [N], y se compara con un umbral para decidir si se ha detectado o no un tono a esa frecuencia determinada. El tamaño de bloque para los datos vocales es de 256 muestras, y se evalúan las muestras correspondientes a las frecuencias DTMF indicadas en la siguiente tabla.

Índice k	*Frecuencia teórica (Hz)*	*Frecuencia real (Hz)*
22	697	687.50
25	770	781.25
27	852	843.75
30	941	937.50
39	1209	1218.75
43	1336	1343.75
47	1477	1468.75
52	1633	1625.00

Tabla 2. Relación frecuencia teórica - frecuencia utilizada.

Puede verse que existe un error entre los valores teóricos de frecuencia a las que debe realizarse la detección y los valores reales que se utilizan. Esto es debido a que las frecuencias evaluadas por una transformada discreta de Fourier son frecuencias discretas que vienen determinadas por la longitud de la secuencia de datos -256 muestras en este caso- y el período de muestreo -125 μs para los canales B a 64 Kbps-. Aun así, el resultado aproximado es lo suficientemente bueno como para que el algoritmo sea útil para detectar la pulsación de teclas.

Se decide que se ha pulsado una tecla si se detectan simultáneamente tonos a una de las frecuencias [697, 770, 852 ,941] y a una de las frecuencias [1209, 1336, 1477, 1633]. Para decidir que se ha detectado una pulsación de tecla, debe ocurrir que se detecte sólo un tono de cada uno de los dos subconjuntos de frecuencias. Si no se detecta ningún tono o se detecta más de uno en cualquiera de los dos subconjuntos de frecuencia, no se decide que se ha detectado una pulsación de tecla. Asimismo, se exige que exista un intervalo de silencio entre dos detecciones de tecla consecutivas, de forma que el algoritmo funcione para ciertos teléfonos que permiten mantener una tecla pulsada enviando los dos tonos correspondientes sin pausa intermedia.

La relación entre frecuencias en Hertzios y teclas es la siguiente:

	1209	*1336*	*1477*	*1633*
697	1	2	3	A
770	4	5	6	B
852	7	8	9	C
941	*	0	#	D

Tabla 3. Teclas del terminal telefónico y frecuencias asociadas.

La última columna de la tabla 3 no se utiliza generalmente, puesto que hay pocos teléfonos que incluyan las teclas A, B, C y D. De cualquier modo, el código realiza la detección de dichas teclas por si en algún momento fuera necesario.

3.6.7. El archivo capifuncs.c.

La última parte de la emulación CAPI consiste en este archivo, que define y exporta las funciones que el usuario debe utilizar para acceder a CAPI. Las más importantes son `capi20_get_message`, `capi20_put_message`, `capi20_waitformessage` y `capi20_get_profile`.

Estas funciones son necesarias para que la biblioteca `c20lib` sea compilable y funcione. Su cometido es transformar la llamada a función correspondiente en operaciones sobre el dispositivo `/dev/capi20`. Por ejemplo, `capi20_get_message` se transforma en una llamada a `read` sobre el dispositivo, `capi20_put_message` en una llamada a `write` y `capi20_waitformessage` a `select`.

Además este código se ocupa de mantener la memoria reservada para los mensajes de datos. En principio se mantienen 7 bloques de 2048 octetos. El número de bloques está tomado directamente de la especificación CAPI 2.0. El tamaño de cada bloque coincide con el tamaño máximo de bloque que la tarjeta AVM A1 puede enviar, y con el tamaño máximo de bloque que utilizaba el controlador original CAPI 2.0 de AVM.

Este archivo debe ser compilado y enlazado estáticamente al programa final, junto con la biblioteca `c20lib`, como luego veremos. Por tanto el archivo `capifuncs.c` no forma parte realmente del controlador de nivel de enlace, pero al actuar como nexo entre el mismo y la aplicación final su interdependencia es tal que me ha parecido conveniente describirlo en esta sección.