CAPÍTULO 4. PORTADO DEL SOFTWARE DE VEHÍCULO AUTÓNOMO AÉREO DE LINUX A QNX.

4.1. Introducción.

4.1.1. Ubicación y funciones del software.

El conjunto de programas que se van a portar van a funcionar en el *PC* de control, sito en el *UAV*. En este capítulo se verá únicamente el portado de Linux a *QNX* del conjunto de códigos de partida del proyecto. Estos códigos serán posteriormente mejorados y ampliados, como se verá en el capítulo 5.

Se describen a continuación las tareas que realizan los programas del software de partida. Para más información, consultar el *Proyecto Fin de Carrera* de David Jareño, "*Diseño para el control automático de un vehículo autónomo aéreo bajo el sistema operativo GNU/Linux*".

Comunicación por puerto serie con el controlador de bajo nivel.

Envío: *waypoints* y órdenes.

Recepción: estado del sistema.

Comunicación por red con el *PC* de visión y con el de tierra.

Control de alto nivel.

Gestión y almacenamiento de la telemetría.

4.1.2. Estructura del software.

El software se compone de tres clases y siete hilos, incluyendo el principal, escritos en once archivos de código y cinco de cabeceras.

4.1.2.1. Clases.

Clase Control:

Esta clase se encarga de gestionar el acceso a las colas, crear la máscara de mensaje (que dirá los parámetros que se necesitan recibir) y codificar (descodificar) la información a enviar (recibida). Cada tipo de dato lleva asociado un buffer en forma de cola circular. A dichas colas tienen acceso todos los hilos del proceso, por lo que es necesario contar con un sistema que permita la lectura y escritura de datos en las colas, garantizando que dos hilos no traten de abrir la misma cola al mismo tiempo. Esta clase contiene una serie de funciones que para cada cola organiza los accesos, de manera que

el manejo de los datos sea seguro. Las normas *POSIX* ofrecen dos posibilidades para este tipo de tareas: los *semáforos* y los *mutex* con o sin *variables de condición*. En este caso resulta más lógico el uso de *mutex* con variables de condición, ya que se trabaja entre hilos de un mismo proceso.

Clase Estimated:

Se trata de una clase muy simple, que incluye funciones para la obtención de los parámetros de la telemetría.

Clase Serial:

La misión de esta clase es la apertura y cierre del puerto serie y la escritura y lectura del mismo. Más adelante se describirá con más detalle el funcionamiento de esta clase, cuando se expliquen los cambios que sobre ella se han hecho, que han sido muchos.

4.1.2.2. Hilos.

Hilo bb:

Este hilo se encarga de las comunicaciones por red con el resto de componentes. Para su labor se apoya en la librería *BBCS*, que ha de estar instalada en el sistema.

Hilo escritura serie:

Ayudado por las clases *Control* y *Serial*, el hilo de escritura serie se encarga de enviar caracteres al *DSP*.

Hilo lectura serie:

Igual que el de lectura, solo que recibe caracteres y los almacena en un buffer.

Hilo movimiento:

Este hilo lleva el controlador de alto nivel del helicóptero.

Hilo pan tilt:

Se encarga del control del *pan & tilt*.

Hilo procesa carácter:

Este hilo "conoce" el protocolo de transmisión, y lo que hace es interpretar los caracteres recibidos para pasarlos a la correspondiente cola, o sacar datos de la misma y generar los caracteres preparados para la transmisión del dato en cuestión.

Hilo main:

Como es obvio, se trata del hilo principal. Es el que lleva el bucle de control y el que lanza el resto de hilos.

4.1.3. Archivos de código.

Códigos de las clases:

bb.cpp
control.cpp
estimated.cpp
serial.cpp

Códigos de los hilos:

hilo_escritura_serie.cpp
hilo_lectura_serie.cpp
hilo_movimiento.cpp
hilo_pan_tilt.cpp
hilo_procesa_caracter.cpp
main.cpp

Códigos de funciones auxiliares:

funciones_main.cpp

4.1.4. Archivos de cabeceras.

colas_uav.hxx:

Descripción: Contiene las definiciones de las estructuras de todos los tipos de colas.

Invocada por: control.hxx.

control.hxx

Descripción: Cabeceras y variables para la clase control.

Invocada por: estimated.hxx.

estimated.hxx.

Descripción: Cabeceras y variables para la clase estimated.

Invocada por: global.hxx, estimated.cpp.

global.hxx.

Descripción: Contiene definiciones para varios usos, y las cabeceras de distintas funciones.

Invocada por: bb.cpp, control.cpp, funciones_main.cpp, hilo_escritura_serie.cpp, hilo_lectura_serie.cpp, hilo_movimiento.cpp, hilo_pan_tilt.cpp, hilo_procesa_caracter.cpp, main.cpp.

serial.hxx.

Descripción: Cabeceras y variables para la clase serial.

Invocada por: global.hxx, serial.cpp.

4.2. Cambios realizados en el portado.

4.2.1. Descripción general.

4.2.1.1. Puerto Serie.

4.2.1.1.1. Introducción.

Como es sabido, el sistema operativo Linux no sigue en su totalidad las normas *POSIX*. Uno de los estándares que no cumple es el del uso del puerto serie. El manejo es muy parecido, y de hecho las estructuras que se usan para la configuración son las mismas. Sin embargo, los significados de los *flags* y de algunos parámetros cambian sensiblemente. Además, la manera en la que *QNX* lee los datos del buffer del puerto serie es diferente a la de Linux, lo cual hace que haya que cambiar parte de la configuración, como más adelante se explicará. Ahora se describe el procedimiento para la inicialización y uso del puerto serie tal y como se necesita en este proyecto, según las normas *POSIX*.

4.2.1.1.2. Apertura.

QNX ve los puertos serie como si se tratara de archivos. Esos "archivos" están en la ruta */dev/serx*, donde *x* es el número natural. Por ejemplo, si el *PC* en uso tiene dos puertos serie, éstos serán */dev/ser1* y */dev/ser2*.

Como se haría para cualquier archivo, lo primero que se ha de hacer para trabajar con él es abrirlo. Para ello se utiliza el comando *open*, que recibe como argumentos el nombre del "archivo"; y una serie de opciones, que en este caso serán los *flags* *O_RDWR* (posibilita la lectura y la escritura) y *O_NOCTTY*, que impide que el sistema remoto pueda actuar como un terminal. *open* devuelve un entero, que será el *descriptor del archivo*. En términos prácticos se utilizará como el puntero a archivo (*FILE **) que necesita el comando *fopen*.

4.2.1.1.3. Inicialización.

Después de abrir el puerto serie, hay que configurar sus parámetros, de manera que el protocolo sea el mismo que el que utiliza el otro aparato de la transmisión, en este caso el *DSP*.

Con la función *tcgetattr* se recogen los atributos actuales que tiene el puerto serie en una estructura del tipo *termios*. Esta orden se ejecuta dos veces. Una de ellas guarda la configuración del puerto en ese momento para restaurarla posteriormente, cuando se acabe de usar el mismo. La otra será la estructura de partida, sobre la que se harán los cambios necesarios. Podría empezarse de cero, con una estructura vacía. Sin embargo es recomendable comenzar con la configuración por defecto, ya que pueden existir parámetros que el usuario desconoce, y que por tanto no puede manipular apropiadamente.

A partir de aquí se procede con el ajuste de los *flags*. Mediante estos señalizadores se le dice al sistema el protocolo a seguir (8 *bits* por *byte*, un *bit* de parada y ninguno de paridad), y se ajustan otras opciones, como la desactivación de los ecos.

Otro campo ajustable es el *c_cc*. Se trata de un vector de dos enteros (*VMIN* y *VTIME*). Al invocar *read* el hilo se quedará bloqueado hasta que se reciban *VMIN bytes* o transcurran *VTIME/10* segundos. En caso de ser *VTIME* cero, el hilo esperará indefinidamente hasta que lleguen los caracteres.

Para configurar las velocidades de entrada y salida se utilizan *setispeed* y *setospeed* respectivamente.

Finalmente se imponen los nuevos atributos al sistema, mediante la orden *tcsetattr*. Aunque no es necesario, resulta conveniente eliminar los datos que haya en el buffer escritos pero no transmitidos o recibidos pero no leídos. Esta tarea la cumple el comando *tcflush*.

4.2.1.1.4. Lectura y escritura.

Las operaciones de lectura y escritura en puerto serie se hacen con los comandos *read* y *write* respectivamente. El primero recibe como argumentos el descriptor de archivo y el tamaño máximo a leer. En este caso ese tamaño va a ser de diez veces el tamaño del mensaje, si bien lo lógico es que cada vez que *read* sea llamado reciba simplemente tantos caracteres como grande sea el mensaje, ya que el número mínimo de caracteres, que viene determinado por *VMIN*, será el propio tamaño del mensaje. Éste es uno de los cambios respecto al software de Linux, donde *VMIN* valía la unidad. En *QNX* esto obligaba a entrar repetidamente en el bucle de lectura, ya que se leían muy pocos caracteres cada vez que se entraba en el buffer; a diferencia de Linux, donde aun con *VMIN* valiendo la unidad, se leía el mensaje entero. En el caso de *write QNX* sí escribe el mensaje entero de cada vez, por lo que *VMIN* vale uno. Para una mayor seguridad, a la función de escritura se le ha añadido un bucle, que fuerza a escribir todos los caracteres

requeridos, salvo que existan errores. También se le ha añadido la función *tcdrain*, que fuerza a que todos los caracteres escritos sean transmitidos.

4.2.1.2. Temporizador.

La versión de Linux que se utilizaba en el *PC-104 (Debian woody)* no tenía cobertura para temporizadores *POSIX*. En la aplicación existía un único temporizador, que cada 10 ms enviaba la señal *SIGALRM*. El hilo principal estaba programado de manera que cada vez que se recogiera dicha señal saltara un manejador, que una por una iba ejecutando las tareas de control. Este temporizador es portable a *QNX*. Sin embargo se decidió que, dado que *QNX* sí tiene soporte para temporizadores *POSIX*, sería conveniente aprovecharlos. Los temporizadores *POSIX* son más fiables, permiten mayor precisión, y además se pueden utilizar varios simultáneamente, sin que interfieran unos entre otros. En efecto, las señales que envían los temporizadores *POSIX* van de *SIGRTMIN* a *SIGRTMAX*, por lo que se tienen varias posibilidades, con diferentes prioridades. Además la señal *SIGALRM* queda libre, y podría utilizarse para otras funciones. Como por ahora se necesita un único temporizador, la prioridad de la señal que envíe es indiferente. Se ha escogido *SIGRTMIN*. En caso de que en una ampliación se necesiten más temporizadores, habrá que plantearse cuáles han de llevar mayor o menos prioridad.

4.2.2. Enumeración de los cambios en cada archivo.

A continuación se exponen los cambios que se han hecho en los códigos del software original (ver *PFC David Jareño "Diseño para el control automático de un vehículo autónomo aéreo bajo el sistema operativo GNU/Linux"*). Para alguien que no conozca el mismo serán difíciles de entender, sobre todo la parte de la clase del puerto serie. En el archivo *serial.cpp* vienen explicados detalladamente los procedimientos de apertura y configuración del puerto serie, y consecuentemente el funcionamiento de la propia clase.

4.2.2.1. Código fuente.

4.2.2.1.1. serial.cpp.

VARIABLES NUEVAS:

Se declara la variable externa *puerto*, que es una cadena de caracteres que deberá contener el nombre del puerto serie.

init_puerto_serie:

Tiene un nuevo argumento, que es el tamaño mínimo de paso.
Se comprueban los parámetros de tamaño mínimo y velocidad.
La velocidad ahora estará en una variable del tipo *speed_t*.
Cambian algunos *flags*. No merece la pena enumerarlos.
El parámetro *c_cc[VMIN]* ya no siempre es 1.

En *open* se cambia el primer parámetro, de *port* a *puerto*. De esta manera los dos parámetros de la clase *serial* quedan obviados. La velocidad lo estaba ya en Linux, y el nombre del puerto lo queda ahora. No obstante se respeta la cabecera del constructor.

inicia_puerto_serie:

Tiene un nuevo argumento, que es el tamaño mínimo de paso.
Llama a *init_puerto_serie* con su nuevo argumento.

enviar_cadena_puerto_serie:

Se le ha puesto un bucle para asegurar que envía todo lo que se le diga.
Se le ha añadido un retorno, que valdrá el número de caracteres escritos.
Detección de errores en la escritura.

recibir_cadena_puerto_serie:

Detección de errores en la lectura.

4.2.2.1.2. hilo_lectura_serie.cpp.

Variables nuevas:

Se añade *extern int tam_msg*

funcion_hilo_lectura_serie:

A *inicia_puerto_serie* se le pone su nuevo argumento a *tam_msg*.
Si *tam_msg > 255* se va reduciendo a la mitad hasta que sea menor, ya que éste será el valor final de *VMIN*, y el mencionado parámetro no puede superar este número.
buffer tiene un tamaño de 2560, que siempre será mayor que *tam_msg*10*, cuyo máximo es 2550.
En *recibir_cadena_puerto_serie* se le cambia el 10000 por *tam_msg*10*.

4.2.2.1.3. hilo_escritura_serie.cpp.

funcion_hilo_escritura_serie:

A *inicia_puerto_serie* se le pone su nuevo argumento.

4.2.2.1.4. funciones_main.cpp.

Cabeceras y nuevas variables:

Se le añade la cabecera *<time.h>*
Se le añade la variable externa *puerto*, que es una cadena de caracteres donde se tendrá que escribir el nombre del puerto serie a utilizar.

configuracion:

Se le añade un "detector de retorno de carro" para evitar que lo lea de *uav.conf* si lo tiene.
Se le han añadido marcadores que sacan por pantalla la configuración.

Al leer el archivo de configuración se añade un último caso, que se corresponde con una línea nueva que se le ha añadido al mismo. En este caso (*case 22*) se lee el número de puerto serie a usar, y se escribe el mismo en la cadena *puerto*.

inicializa_timer:

Se cambia por completo para adaptarla a las normas *POSIX*. Ahora usará *SIGRTMIN*, y no *SIGALRM*.

4.2.2.1.5. main.cpp.

VARIABLES NUEVAS:

int tam_msg. Dirá el tamaño de la cadena de caracteres a recibir. Será el que se use en la inicialización del puerto serie (parámetro *VMIN*).
char puerto[20]. Se inicializa a */dev/ser2*. Esta variable global dice el puerto serie que se va a utilizar.

main:

Se calcula *tam_msg* mediante la función *p_control->calcula_tamano_total(datos_configuracion.mascara)* antes de inicializar el puerto serie.

envio_periodico:

Se le ponen los argumentos que necesita un manejador para el parámetro *sa_sigaction* de una estructura *sigaction*. Es decir: (*int signo, siginfo_t* info, void* other*).

4.2.2.1.6. control.cpp.

VARIABLES NUEVAS:

pthread_cond_t loop_cond_var_2=PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex_wait_2=PTHREAD_MUTEX_INITIALIZER;
extern int tam_msg, variable que dirá el tamaño de la cadena de caracteres a recibir. Será el que se use en la inicialización del puerto serie (parámetro *VMIN*).

calcula_tamano_total:

Función nueva. Función pública que calcula el tamaño de la cadena a recibir, incluidos los bytes de inicio y cola.

espera_fin_bucle:

Se han cambiado el *mutex* y la variable de condición del *pthread_cond_wait* de *loop_cond_var* a *loop_cond_var_2* y de *mutex_wait* a *mutex_wait_2*.
 Detección de errores en *mutex*.

fin_bucle:

Se ha cambiado la variable de condición del *pthread_cond_signal* de *loop_cond_var* a *loop_cond_var_2*.

4.2.2.2. Archivos de cabeceras.

serial.hxx

Cambio de los nombres del puerto serie de Linux (*/dev/ttyS**) a *QNX* (*/dev/ser**). *init_puerto_serie* e *inicia_puerto_serie* tienen un nuevo argumento, que es el tamaño mínimo de paso.

Se crea la variable privada *vmin*, que será la que le dé valor a *c_cc[VMIN]*.

Se cambia la cabecera de *envia_cadena_puerto_serie*, ya que ahora devuelve un entero, y no *void*.

global.hxx

Se cambia la cabecera de *envio_periodico* para ponerle sus nuevos argumentos.

control.hxx

Se le añade la cabecera de la función pública *void calcula_tamano_total(int mascara);*

4.2.2.3. Makefile.

LDLIBS

Se ha eliminado *-lpthread*, ya que en *QNX* las funciones para los hilos vienen en la librería estándar (*libc.so*).

Se ha añadido *-lsocket*, pues *QNX* necesita de esta librería para trabajar con los distintos protocolos de red.

CFLAGS

Se ha añadido *-D QNX*. Esta opción de compilación define la macro "*QNX*". Así es posible tener los códigos de Linux y *QNX* en los mismos archivos.

4.2.2.4. uav.conf.

Se ha añadido una última línea, donde se debe escribir el número de puerto serie a utilizar, empezando por el 1. Es decir, *1=/dev/ser1*, *2=/dev/ser2*, etc.

4.3. Problemas con el puerto serie.

Aunque del portado de los códigos para el puerto serie ya se habló anteriormente, conviene hacer un nuevo apartado para explicar los problemas surgidos a la hora de ejecutar el programa en el *PC-104*. En efecto, tras hacer la programación del puerto serie, y comprobar su funcionamiento en el *PC* de mesa, se procedió a la instalación del programa en el *PC-104*. De manera sorpresiva, éste resultó fallar, provocando un error de segmentación que hacía caer la aplicación.

Este error de segmentación se debía a un fallo de programación, que fallaba al recibir caracteres extraños por el puerto serie (ver capítulo 4, apartado 4.2.1.2.). Tras su corrección se hicieron una serie de pruebas en las que se comprobó que la aplicación

perdía aproximadamente entre el dos y el tres por ciento de los paquetes de caracteres que le eran enviados. Curiosamente este problema sólo aparecía cuando se llamaba al ejecutable desde el *shell*. La aplicación funcionaba a la perfección cuando era invocada desde un terminal remoto, con *telnet*.

Al principio se pensó en un error de instalación. Para comprobarlo, se arrancó el sistema desde el *CD-ROM*. Se ejecutó la aplicación, y ésta recibía los mensajes sin problemas aparentes.

La siguiente prueba fue instalar *QNX* desde el *CD-ROM*. Ya se explicó en el capítulo anterior que esto conllevaba una serie de problemas. Los resultados de la prueba fueron aún más sorprendentes que los que ya se tenían. El programa recibía sin problemas los mensajes cuando la interfaz gráfica estaba arrancada, pero volvía a tener exactamente los mismos problemas que al principio cuando no lo estaba.

El siguiente paso fue probar en el *PC* de mesa, con y sin interfaz gráfica. En ambos casos la recepción era correcta.

Por último se decidió instalar el disco duro del *PC* de mesa en el *PC-104*, y probar la transferencia. El resultado fue idéntico al de la instalación desde *CD-ROM*: en modo gráfico no hubo problemas, y en modo comando perdía paquetes.

Se llegó incluso a instalar una imagen creada a partir de *QNX* 6.3.0 (en vez de 6.2.1), que tampoco resolvió el problema.

Después de todos los intentos anteriores se llegó a la conclusión de que el problema no estaba en la programación. Este tema se preguntó en los foros de *QNX*, de donde no sólo no salió una explicación, sino que otros usuarios del mismo tipo de *PC* (*Advantech PCM-3350*) reconocieron haber tenido otros problemas con los puertos serie de la máquina, con soluciones en algunos casos verdaderamente ilógicas (ver anexo 6).

El *PC-104* irá dentro de la caja del hardware del helicóptero, por lo que la única manera de acceder a su software es mediante *telnet*. Todas las pruebas en las que se ha llamado a la aplicación desde otra máquina han sido satisfactorias, por lo que ésta no debe perder caracteres cuando se ejecute en un caso real. No obstante, en caso de una recepción incorrecta la cantidad de paquetes que pierde no es excesiva, por lo que el programa seguiría funcionando sin mayor dificultad.

En esta tesitura, y tras haber arreglado el *bug* que hacía fallar al programa (capítulo 5, apartado 5.2.1.2.), se entiende que no resulta productivo continuar buscando una solución, que posiblemente no exista, a un problema que aunque se diera tampoco iba a ser relevante.

4.4. Conclusiones.

Se ha comprobado que en general el portado de software de Linux a *QNX* no proporciona grandes problemas. Lo único que cambia de un *SO* a otro son las librerías. Pero, como es lógico, las funciones estándar de *C* y *C++* siguen siendo las mismas.

Sin embargo para este programa en particular sí han habido algunas dificultades. La primera ha sido el uso de las librerías que implementan las normas *POSIX*. Si *QNX* cumple prácticamente todas estas normas, Linux no lo hace, por lo que se tienen que realizar ciertas variaciones en el código, no siempre triviales.

Por otra parte están los problemas con el puerto serie; tanto los de hardware, relatados en el apartado 4.3, como los de software, también debidos al incumplimiento de ciertas normas *POSIX* por parte de Linux; así como a las diferencias de ambos sistemas a la hora de leer el puerto.

Finalmente la falta de estructuración del programa original, y sobre todo la gran escasez de comentarios en el mismo han hecho muy complicada la comprensión del código, y con ello del portado.

Las aplicaciones para las que el programa está diseñado no necesitan un control estricto del tiempo real. Esto hace que no se noten las diferencias entre el uso de Linux y *QNX*. Aunque *QNX* en teoría debe ir mejor, resulta complicado hacer una comparativa, ya que en la práctica ambos sistemas operativos cumplen sobradamente las especificaciones necesarias.