KNX Implementation for Plan 9
Work In Progress

Gorka Guardiola Múzquiz
Enrique Soriano Salvador
Francisco J Ballesteros

Laboratorio de Sistemas
Universidad Rey Juan Carlos

ABSTRACT

We are working on a project to interconnect and control home devices from a hive of mini computers. We have started with sensors and actuators for the home. In order to do this we have written an implementation of the Knx [1] protocol stack for Plan 9. We plan to use gumstix computers and therefore we are porting Plan 9 to this platform.

Introduction

Nowadays homes are full of devices begging to be interconnected and controlled. Big screens, audio devices, computers and computer peripherals, switches, lights, sensors for temperature and movement, alarms and possibly many others depending on the tastes and economic possibilities of the owner. We envision a network interconnecting these devices and a computer controlling them to make them an integral system which gives the user a seamless experience.

We are working on a project to export these devices as filesystems using small minicomputers. As part of this project we have started a port of Plan 9 to the gumstix (a small arm pxa270 based minicomputer) and written a port of the Knx protocol stack to Plan 9. This article describes the Knx usb driver we implemented and the development of a Knx protocol stack under Plan 9, and its filesystem interface. The filesystem interface is still under development.

Knx

Knx is the successor of EIB, European Installation Bus. Knx lets you connect a network of sensors, actuators and small devices using a Knx bus, which defines various physical mediums including a dedicated cable, and an insanely complicated protocol. Knx is an standard supported by many companies and it is easy to find sensors and actuators which have been reported to be quite robust and which consume very low power.

We have finished the Knx protocol development and we are now working on a good interface to export the filesystem.

Most of the available devices to control Knx networks just export the network through a gateway which lets you inject packets into the bus. This approach just translate the difficulties of programming the Knx protocol adding another layer without abstracting the problem itself.

We plan to export a synthetic filesystem in the Plan 9 tradition, which will make it much more simple to control the devices. Also, the devices themselves can run many different "applications", because they can be programmed in some metalanguage

combined with assembler. We want to control Knx devices as much as possible, but we are not particularly concerned in programming the devices themselves, just with configuring and controlling them.

The devices can be programmed using the standard tools to program them with the default binaries provided by the vendors as we have intelligence in the computer to do the rest for us. Once programmed, the devices offer "objects", data types with a network address which can be accessed through the devices or through their special object address. We plan to have unique object addresses throughout the device and the network if possible. This way, we can control each device and its properties from a computer (the gumstix or the client to its file server).

Knx USB device

Knx defines a protocol for a USB [2] "coupler" to connect to the bus and act as a bridge between the bus and a PC. The first thing we had to do was write a driver to control the "coupler" itself.

The Knx USB devices provides us with two 64 bytes interrupt endpoints besides the control endpoint, used for communicating with the device. The devices announces itself as an HID [3] device and uses HID report headers and bodies format. The sequence low nibble is used for multisequence packets, but we have not found a device for which we needed to implement this.

The USB device has two parts, the Bus Access Server (Features) and (possibly) various Emi Servers. Emi, external message interface is the name of the protocol as seen outside of the bus. The Knx messages have two representations internal (Imi) and external (Emi) and the Emi Servers translate between one and the other. The Emi Servers are more than that, they filter and rewrite incoming packets depending on their layer they are configured to understand and the address of the packet. Essentially they implement all the protocol stack. There are two kind of packets which we send to the USB device in the report body. Packets directed to the Bus Access Server, which decides which Emi Server to turn on and how to configure it, and Emi packets which are intended to be sent to the BUS.

Knx network protocol

Knx uses the ISO request, confirmation, indication structure for its communications. Each time a request message is sent the sender receives a confirmation from his local Emi Server and the receiver receives an indication. We have ignored confirmations as they are generated locally (so they do not confirm anything) and when we are waiting for a response we always set a timeout, in some cases defined by the standard and in many other cases just a sensible limit. There are also retransmissions, so there is no point in confirmations at all.

The Knx standard defines a the whole link/transport/network layer and modes for each of this layers (and extra routing properties) for the Emi Server. We configured the Emi Server in the (link layer, bus monitor), which is an essentially transparent mode so that we could have complete control and see what was exactly happening with the devices themselves. Even in this mode connected communications between remote devices are not seen, which makes it difficult to program a real sniffer (other than continuously changing address). In any case, at least using the lowest possible mode lets us do broadcast to program an address an in general program operations without having to change modes continuously at the risk of creating a race condition and losing packets. When the Emi Server is configured in an higher layer the packets arrive with the upper part erased (zeros) or do not arrive at all.

As a consequence of having the device configured at the lower layer, we see broadcast packets, and other packets we could ignore (confirmations), and we have to deal ourselves with retransmissions and timeouts.

The Knx stack

We created two I/O procs to take care of the blocking I/O of the USB endpoints. The rest of the protocol stack is composed of threads all living in the same proc. `Usbwriter` and `usbreader` are threads representing the I/O procs and which serve a channel each. These threads convert the raw USB packets int structs, taking care of the HID report headers and the Knx transport headers. This interface permits the stack to send Bus Access Feature packages whenever needed and inject them at low level while at the same time not having race conditions with other layers of the stack.

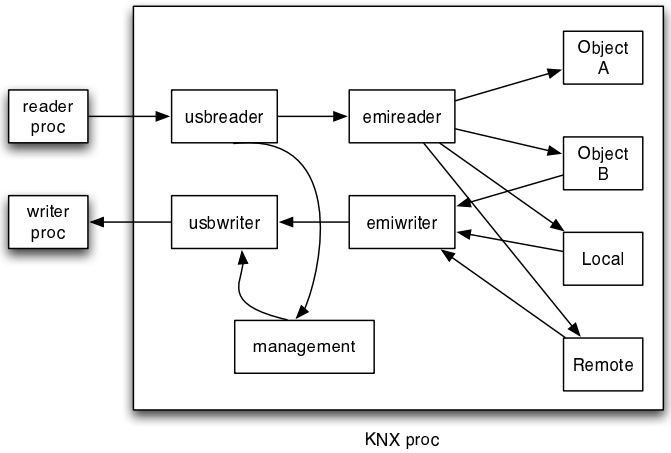

Figure 1: Architecture of the Knx driver.

Two other threads, `emireader` and `emiwriter` read from these channels, decode and interpret the packets and send them to the appropriate thread. There is an another thread representing each addressable Knx object plus three extra threads, **Management**, **Local** and Remote. **Management** is the thread representing raw non-emi packets, essentially Bus Access Features. **Local** injects emi packets in raw, normally used to configure parts of the Emi Server. **Remote** is used for injecting raw Emi packages into the bus for things like configuring devices which have no address yet.

We have implemented stop and wait, taking care of retransmissions as part of the sending functions, to make things simpler. At the moment, there is no need to optimize throughput in any case.

Current state and Future work

The protocol stack as it is lets us already program addresses for the devices and control all the devices we have. We can read the state of switches, control binary outputs and read the state of temperature and light sensors. The Gumstix port is already working. The only work left is some minor fixes and device drivers. The ethernet device driver is working in Inferno and has to be ported to Plan 9.

We cannot (yet) discover the objects within a device we have not programmed ourselves. We cannot program Knx devices, though this was never our goal it would be nice to be able to write a binary image on them to program them. A filesystem to control the objects in under way. We plan to have a directory representing each of the object threads, though we are working on the how many files it will contain and their

semantics.

References

1. K. Association, KNX Association Official Website, http://www.knx.org.
2. Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC and Philips, USB 2.0 Specification, 2000.
3. U. I. Forum, USB Device Class Definition for Human Interface Devices (HID) , 2001.