# Data Transport on the Networked Surface

Frank Hoffmann, James Scott † Mike Addlesee, Glenford Mapp * Andy Hopper †*

Laboratory for Communications Engineering † AT&T Laboratories Cambridge *
University of Cambridge

E-mail: [fh215,jws22]@cam.ac.uk [mda,gem,hopper]@uk.research.att.com

## Abstract

*Networked Surfaces are surfaces such as desks which provide network connectivity to specially augmented devices, for example handheld computers. When the devices are physically placed on top of the surface, they can connect to different kinds of services – mainly, but not exclusively to send and receive data.*

*This paper discusses challenges in implementing Networked Surfaces, paying particular attention to data flow issues, focusing on how the various software and hardware entities comprising the Surface interact to transport data to and from objects.*

*Keywords: Mobile Networking, Sentient Computing*

## 1. Introduction

This paper describes the various software and hardware entities that make up the Networked Surface, concentrating on how these entities interact to achieve data transport between the Internet and devices on the Surface.

In the first section, a short overview of the concept of Networked Surfaces will be presented, and comparisons to wired and wireless networks will be drawn. Other services that can be provided, such as location information for objects, will also be highlighted.

The second section will discuss the process of connecting a device to the Networked Surface busses, in order to allow data paths to be established. Details are also given of the different types of network provided, to support the various classes of devices, from palmtop to PC.

The third section looks at implementation details of the "Surface Manager", which bridges data between conventional networks and Networked Surface busses. Hardware, kernel-level software, and user-level software must all interact to enable this data flow.

The fourth section discusses and interprets preliminary measurements of how long it takes to connect an object as well as of the data rate and the reliability of a data channel once the object is connected.

### 1.1. Wired vs. Wireless

The world of today contains more and more electronic devices that can store and exchange data in private and professional life.

Desktop computers are common in every office, more laptops are in use and in addition an increasing number of smaller devices such as digital cameras, personal digital assistants (PDA), players for digitally stored music and mobile phones are more commonplace. All these devices need to exchange data to be useful. For example, users will want to download music from their desktop into their MP3 player or store minutes from a business meeting from their PDA onto a laptop, etc..

Desktops are typically connected to high speed ethernet networks and laptops are increasingly connected to wireless ethernet, representing examples of the main two ways to connect devices, namely *wired* and *wireless*.

Wired connections offer high bandwidth as in the case of ethernet. They also offer a wide range of different types of connections including USB and RS-232, which are in wide use for the smaller devices mentioned. Unfortunately, with more and more of these devices, the users also get more and more different cables, cluttering desks.

Wireless networks such as wireless ethernet [5] or BlueTooth [1] offer a solution to the problem of too many wires, but typically provides less bandwidth which, in addition, is inherently shared. This disadvantage is likely to be amplified as more of the smaller devices gain wireless connectivity.

### 1.2. The Networked Surfaces Concept

The Networked Surfaces project represents an attempt to keep the advantages of both methods of connectivity while minimising the disadvantages. The goal is to be able to connect various devices to a network that fulfills their needs in

terms of bandwidth (that is, higher bandwidth for devices like laptops and lower bandwidth for devices like PDAs), in a manner which is intuitive to the user, eliminating the need to know which device connects with which cable and avoiding tedious setup procedures of multiple serial ports etc.

In order to achieve these goals, it is proposed that devices be connected by simply placing them onto a special surface such as desks. Both the surface and the device are augmented to enable them to recognize each other and to negotiate what types of connections that particular object needs, and whether they are available on that particular surface, enabling us to use the same intuitive connection method for different devices. The surface is permanently connected to an entity called the *Surface Manager*, which serves the dual purpose of managing the surface itself, as well as providing the means for various objects to send and receive their data.

Thus, with this method, devices can be connected while they are placed on the surface. While this approach does not offer quite the same freedom of movement that is provided with wireless networks, it is sufficient for most purposes: digital cameras for example typically do not need to take pictures while they are downloading their contents onto another computer, users do not listen to music while they are downloading it into an MP3 player and laptops are commonly used for extended periods of time while they are placed on desks.

## 1.3. Additional Advantages

In addition to data transmission, the proposed design can also offer other useful services to devices.

Since portable devices typically only draw a small amount of power from their batteries, it is possible to provide power as an additional service on the surface, given that appropriate safety measures are provided. This way the object would not need to use its batteries while it is placed on the surface. Furthermore, the surface could also provide energy to charge the object's battery. Mobile phones, for example, could easily be recharged this way, since they are often taken out of pockets and placed on desks.

Furthermore, *location information* for every object placed on the surface can be provided, a very important quantity for sentient computing [8, 3], which in turn enhances and further simplifies the use of devices.

For example, instead of directly connecting the digital camera to a specific workstation via a cable, location information could be used to infer that connection. Given that the location of the workstation and its keyboard is known, the user could specify to download the pictures from the camera to that particular desktop computer by placing the camera close to the keyboard on a networked surface.

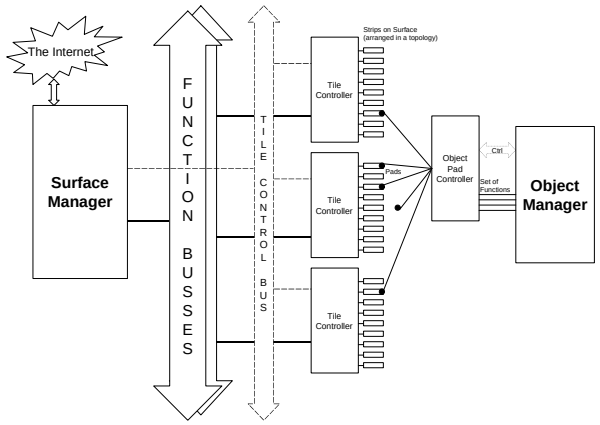


**Figure 1. Networked Surface Architecture**

## 1.4. Design Overview

Only a brief overview of the design can be presented here, but a more detailed description can be found in [9].

The described architecture is illustrated in Figure 1.

**1.4.1. Topology.** In the current prototype, the connection of objects to the surface is achieved by direct touch of metal contacts[1] placed in a certain *topology*.

On the surface, the contacts have the form of *strips* organized in columns, while on the objects they are represented in small round *pads* placed in a circle, as shown in Figure 2. In order to support a given type of network, a certain number of connections are required (for example a bidirectional serial connection requires three connections for ground, transmit and receive), resulting in a smaller or bigger circle in which the pads are placed for different devices.

**1.4.2. Scalability through Tiling.** To monitor each of the strips of the surface a certain amount of electronics is needed. Implementing this circuitry in one centralised piece of hardware would make the system very inflexible to changes. Hence to keep the system scalable, the surface is divided up into smaller parts called *tiles*, controlled by *tile controllers*.

Each tile controller includes a small microprocessor to monitor the strips and detect objects autonomously, as well as circuitry for each strip to send and receive a low bit-rate signal, necessary for the process of detecting devices and determining the type of connection they need.

[1] The object and the surface could also be connected in other ways, using capacitive [10] or inductive coupling, but while it is planned to address these possibilities later, it is not within the scope of this paper.

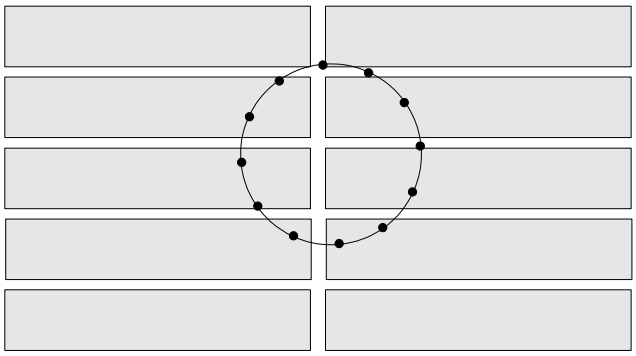

**Figure 2. Example Layout for Chosen Topology**

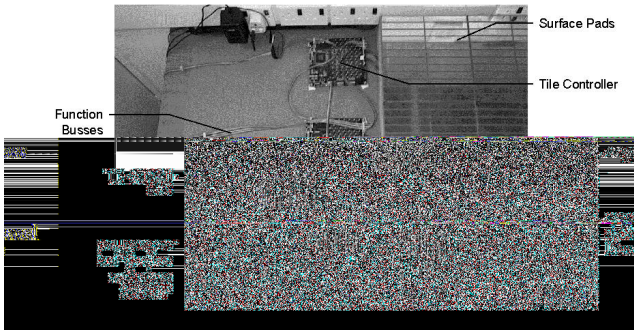


**Figure 3. Networked Surfaces Prototype**

A control bus permanently connects all tiles to the surface manager. This bus is implemented using $I^2C$ bus, a standard originally developed by Philips [4]. $I^2C$ is a synchronous, multi-drop, wired-and bus and was chosen, because it specifies bus arbitration as well as framing of packets and is readily available in many microprocessors.

Using the control bus, the surface manager controls the tiles by sending messages concerning the connection and disconnection of strips, as well as the synchronization of the tiles, which is important for the process used by the tiles to detect new objects.

**1.4.3. The Data Busses.** All the data busses are connected from the manager to each tile, enabling the tiles to connect objects to the functions they desire and managing the data traffic from and to the devices.

Since multiple objects might share a data bus, the busses inside the surface must be multi-drop busses. Currently two types of data connections are implemented. For low bitrate devices an $I^2C$ bus is used again, while for high bitrate devices, B-LVDS [2] is used. B-LVDS uses differential signalling to decrease the sensitivity to noise coupling into the signal. LVDS was chosen, because it only specifies the physical signal levels, hence giving the opportunity to compare different framing and arbitration schemes.

In order to offer more bandwidth, multiple instances of this bus are implemented, which also allows a limited way to guarantee quality of service for devices.

**1.4.4. The Object.** For the object, the same system components need to be replicated. The object only has one set of pads, so no tiling is necessary. Hence the *object manager*, which gives the object access to the data bus and exchanges control messages with its counterpart on the surface, might be implemented in the same unit as the *object pad controller*, the component which monitors the object's pads.

Figure 3 shows a photo of the implemented prototype, showing the surface tiles and their controllers, as well as object hardware performing the functions of both controller and manager. The surface manager PCI card is also shown.

## 2. Networked Surface Objects

### 2.1. Connection and Disconnection

One of the novel and interesting problems in implementing Networked Surfaces lies with the highly dynamic nature of the network topology. Connection and disconnection can occur at any time, causing an object to be added or removed from the network.

It is important to connect objects to the network as quickly as possible, in order to reduce the latency between the time the object touches the surface and the start of data transfer. For example, a user placing a PDA on the surface should not be kept waiting long before their PDA could start "synchronising".

Less obvious is the importance of disconnection speed. Quick disconnection detection allows the surface pads used by an object to be made quickly available for another connection. One particular case where this is important is if a connected object is moved slightly, so that the object still spans many of the same surface pads, but is moved sufficiently that the object-to-surface-pad mapping is no longer valid. In this case, the time-to-reconnection for the object depends on both the connection delay and the disconnection delay so that the pads are available for reconnection. One example of this would be a user with a notebook computer, who moves it slightly to show a colleague some results on the screen, but would not expect to lose their network connection for a significant amount of time.

**2.1.1. Connection Details.** As previously mentioned, the networked surface consists of many tiles connected to a single surface manager. The use of many tiles promotes scala-

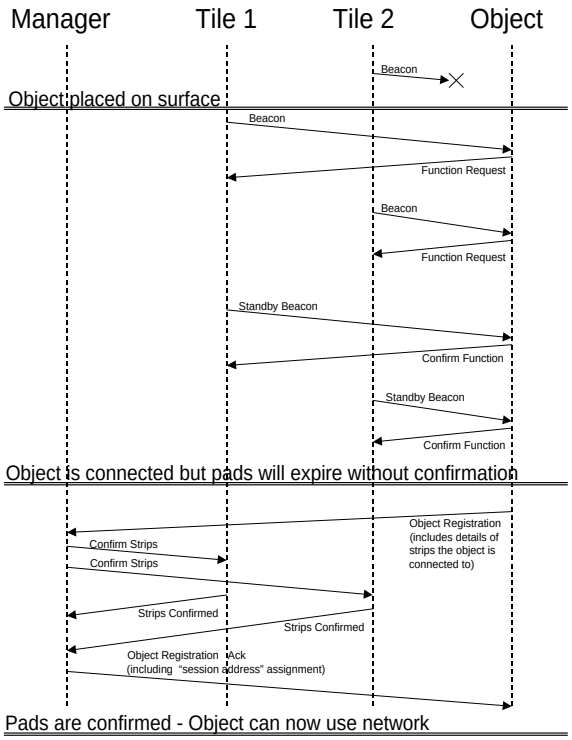


**Figure 4. Handshaking and Registration Protocols**

bility, as each tile is responsible for handshaking with new objects for its own portion of the surface, and the surface manager does not need to directly control each surface pad. Figure 4 shows the handshaking protocol.

In order to handshake, the tiles each send "beacon" messages on each surface strip periodically, cycling through each strip in turn. When an object detects a beacon, it "asks" for one of the functions it requires, which the tile then marks as in "standby" mode. In the standby mode, the tile sends modified "standby beacons" periodically on the strip, but these beacons specify which function was asked for, so other objects receiving this beacon will not attempt to use the same strip. After the object has reserved one pad for each function it requires, it starts responding to "standby" beacons with a "connect" message, and the tile and object both connect the pad to the appropriate function.

This procedure sounds suitable for the connection process as-is, but it is actually flawed in that there is no way for disconnection to be detected, since the tiles do not have the capability to monitor the function busses. A connected pad would therefore remain so indefinitely. Therefore, the first thing that an object must do after it is connected, is to notify the Surface Manager (using its newly acquired network) of its existence and also informing it about the tiles and strips on those tiles through which it is in contact.

Furthermore, to guard against a situation where tile pads are connected but the surface manager is not informed, due to bus error, disconnection, a malicious object, or other means, the tiles are made to time out connected strips and return them to a handshaking state, unless a connection is "confirmed" by the surface manager on those strips. In this way, the connections process is made robust - the surface manager must explicitly take responsibility for an object before the strips it is using are confirmed.

**2.1.2. Disconnection Details.** In general, disconnection can be detected in one of two ways. Firstly, it can be detected using hardware-based methods, in cases where disconnection results in different behaviour to a merely idle state. For example, an incoming power line may no longer be driven. Secondly, disconnection can be detected using "soft" methods over a network, whereby "ping" messages would be sent if no communication occurred over a set time, and disconnection would be detected if the ping were not replied to.

While the first method is by definition faster (it results in an asynchronous notification as soon as disconnection occurs), on the surface it is not possible to use this method since the surface manager is potentially providing at any time for many objects all sharing access to function busses, and it would not be able to infer the disconnection of any particular object from the bus behaviours. Therefore, on the surface side, the "soft" approach must be taken. However, this does not impose overhead on the networks, since "pinging" only need occur if the object were otherwise idle; if the object were using the network, the user-level object registrar could simply spy on the packets passing and note that that object is still connected.

On the object side, disconnection detection could be achieved using either method, and in particular if an object were requesting power as a function, this would be a very good candidate for hardware-based disconnection detection.

## 2.2. Different Networks for Different Devices

It is the aim of the Networked Surface physical layer, including the tiles and object controllers, to provide "generic wires" to the layer above, i.e. to provide physical transmission links which are generic, in the same way that a bundle of wires might be used as a phone cable, an ethernet cable, or a power line. The handshaking process exists to ensure that the correct surface wires, the "function busses", are connected to the correct object wires. This is a very

powerful feature, allowing, for example, re-use of exactly the same technology for a completely different set of services, for example a user's desk might require a USB-like peripheral bus, whereas a conference room table might use the generic wires for an inter-computer network to connect the users' notebook computers.

However, supporting these disparate networks raises problems, both at the link layer and above. Firstly, the surface manager's object registration daemon must be able to receive messages from objects on all available networks, otherwise objects will not be able to register and their pads will time out. Secondly, the surface manager must be able to communicate on each of the networks it provides, and must have the capacity to simultaneously handle traffic on all these networks. Finally, much of the traffic will not be for the surface manager, so the surface manager must act as gateway router for all objects on the surface.

In order to provide for the various possible function busses, two classes of functionality can be identified: IP-capable and non-IP-capable. The former would be for, say, notebook computers using the surface as a means for accessing the internet. The latter would include PDA's which establish serial connections in order to synchronise their contents. One interesting feature is that whether a particular bus is marked IP-capable or non-IP-capable need not necessarily depend on the physical bus type. For instance, RS-232 is commonly used for PDA synchronisation, but it is also used with SLIP as a link layer for IP.

For IP-capable networks, traffic must be transported to and from the IP stack on the surface manager, which will take care of the necessary routing.

For non-IP-capable links. the object simply wants to be connected directly to a particular application which "knows" how to talk to it. In order to achieve this, the object's data stream can be made to appear at a "virtual serial port" on the surface manager. Alternatively, by using IP tunneling, the data stream could appear as a virtual serial port on another machine (for example, the physically closest workstation to the object).

## 3. Surface Manager Implementation

The surface manager is implemented as a Linux personal computer containing a custom PCI card, a software device driver and a user-level software daemon.

### 3.1. Hardware

A custom PCI card is responsible for sending and receiving the data to and from the surface, via the control bus to the tiles as well as via the various data busses. A block diagram of the PCI card can be seen in Figure 5.

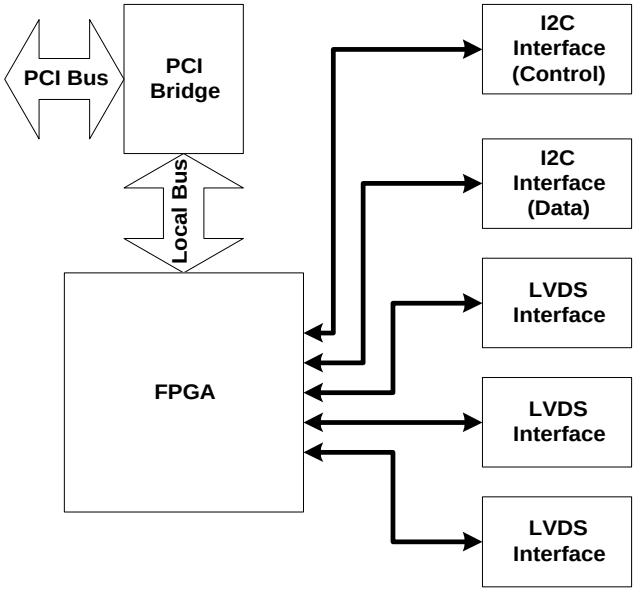


**Figure 5. PCI Card Block Diagram**

**3.1.1. PCI Card Overview.** In order to be addressable on the PCI bus, a bridge chip is used, which translates PCI bus accesses into accesses on a local bus. The local bus confirms to the i960 bus standard [6].

The local bus is mastered by the bridge chip and has one other participant, a field programmable gate array (FPGA). This FPGA contains the logic necessary to interface with the local bus and, more importantly, one "module" for each implemented bus in the surface[2]. The FPGA configuration is implemented in Verilog, a high level description language for programmable logic.

Using a dedicated bridge chip in combination with an FPGA has the two advantages of offering a reliable implementation for the PCI bus on one side, while at the same time giving maximum flexibility for the implementation of the logic controlling the busses on the surface. The latter point is vital to be able to research and compare different coding and framing methods for the LVDS busses as well as being able to do other measurements.

The FPGA also connects to the components to drive the different busses. In the case of the $I^2C$ bus this circuit only consists of two pull-up resistors, since the $I^2C$ protocol is entirely implemented inside the FPGA. In the case of the LVDS busses, one transceiver chip per bus is needed.

**3.1.2. FPGA Modules.** The FPGA decodes a certain range of the i960 address space and subsequently divides this ad-

[2] N.B. We use "module" to describe the control logic for one bus, as distinct from the Verilog keyword "module", which is similar to a procedure declaration.

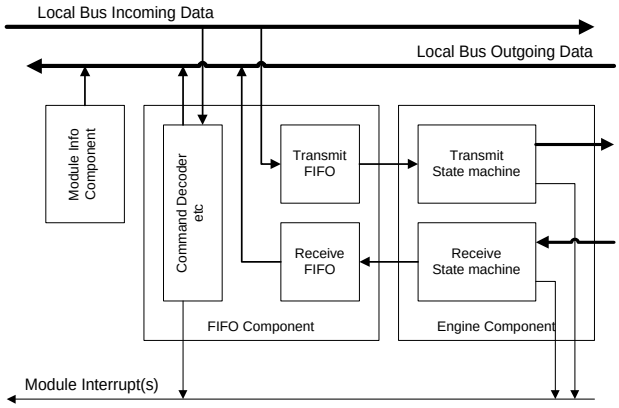


**Figure 6. FPGA Module Internal Architecture**

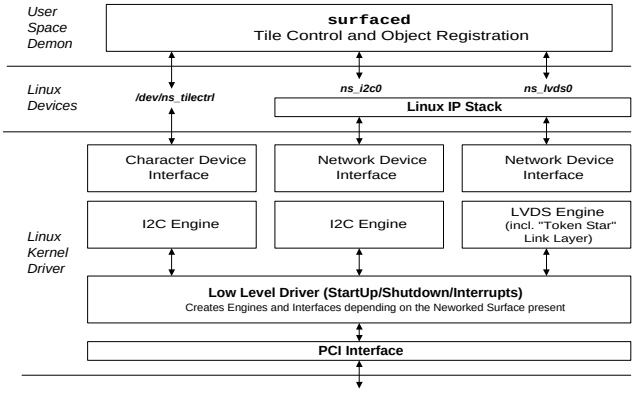


**Figure 7. Surface Manager Software Architecture**

dress range further into "pages" to access the various modules; up to eight modules can be addressed inside the FPGA.

In order to allow an efficient implementation of the software driver for the PCI card, all the modules inside the FPGA follow the same layout.

Each module consists of three components. A block diagram of a typical module is shown in Figure 6.

The first component always consists of a small read only memory (ROM), containing up to eight bytes of data. This ROM describes the type of the module, including additional parameters. If no module is present in a particular page, this ROM reads as zeros. Hence, by reading the ROM at the beginning of each page, the driver can determine which modules are currently implemented inside the FPGA and therefore what functions that particular surface provides.

The second component of the modules that transmit and receive data is occupied by two first-in-first-out (FIFO) buffers, one for each data direction.

This component includes the addresses to access the memories as well as logic to decode necessary commands related to the FIFOs, for example to clear them, or to read how much space is left in either of the FIFOs. Also, the FIFO component has the ability to generate interrupts depending on how full or empty the buffers are, and includes logic to enable or disable these interrupts as well as to specify the levels at which the interrupts are generated.

The third component, called the "engine" component, generally consists of one or more state machines, responsible for sending and receiving the data on the particular bus. Depending on the particular bus type this component varies widely in complexity.

As explained, the data transmitting modules can generate multiple interrupts. However, our implementation allows only a single interrupt to be raised on the PCI bus through the PCI bridge, and hence a special system module is needed, to combine the interrupts from all the other modules. This module deviates from the described layout. It does not require any FIFOs and so, while the first component again describes this module's type, its second component provides the capability to mask interrupts from complete modules in a centralized place.

Using the modules described above, a minimal FPGA configuration consists of one interrupt module, one $I^2C$ module for the surface control bus, another $I^2C$ module as data bus for low bit rate devices and one LVDS-module for high bit rate devices.

## 3.2. Software

This section covers the implementation of software-based components of the Networked Surfaces prototype, namely a Linux device driver, a Linux user-level daemon. The link-layer arbitration and addressing (both software-implemented) are then presented in further detail.

**3.2.1. Flexible Driver.** The surface manager software is in some ways a mirror image of the surface manager hardware. Whereas the PCI card's function is to take every type of incoming traffic and multiplex it onto the internal PCI bus, the driver software's function is to take that generic data, and deliver it to the appropriate destination. Figure 7 shows a block diagram of the software.

In keeping with the "generic wires" concept, the surface manager software is designed to cope with any type of networked surface. Instead of allocating data structures for the precise networks on the prototype PCI card, the device driver dynamically detects the networks available and sets up structures based on that information, which is provided in the "ROM" component of each module in the PCI card. The code and data structures that communicate with each module type ($I^2C$, LVDS, etc) is termed an "engine" to mir-

ror the like-named component in the PCI FPGA, with which it communicates.

Finally, there is the question of where the surface data is transmitted to or received from. This is related to the issue of whether a bus is of the IP-capable or non-IP-capable network type mentioned previously. These categories map neatly into Linux "network" and "character" device categories, and data structures for managing the appropriate one of these "destinations" are instantiated for each bus on the surface (the choice again being dictated by the PCI ROM "bus type" field.

**3.2.2. Object Registration and Tile Control.** In addition to the data path outlined above, the surface manager must provide control functions for the surface, namely for tile control and for object registration purposes. While this could also be done in kernel space, alongside the data transfer functions of the device driver, a different solution is to use a user-space daemon for this purpose, which has been termed `surfaced` to be in line with standard Linux naming (e.g. `nfsd`, `inetd`). This solution is "safer", as the crashing of a user-space program does not bring the system to a halt, memory leaks can be recovered by closing the program, and Linux can pre-emptively multitask the daemon with other applications.

For tile control functions, `surfaced` simply uses the PCI card's tile control bus, which is presented as a Linux character device, to send and receive control messages from the tiles. For object registration, `surfaced` uses ioctl's to bypass the Linux networking stack when communicating with IP-capable objects, and it communicates directly with non-IP-capable objects using character devices. Further work may involve the construction of a UDP/IP interface to `surfaced`.

**3.2.3. Dynamic Addressing.** In order for any object to be used on any Networked Surface, a globally unique address will be required, such as the 48-bit ethernet MAC address. However, using an address of this length in every packet is not efficient, when considering how many objects one could conceivably put on a single networked surface.

Because objects are registered with the Surface Manager immediately on connection, this registration process can be used to assign a short "session address" to objects, which they would use and respond to during that period of connection. In the prototype system, an 8-bit address is used. The object still needs a globally unique IP address for communication with Internet hosts.

When an object is connected to an IP-capable network, the surface manager updates its IP routing tables in order to provide routing services for that object.

For objects moving round in a single subnet, the surface manager can use "proxy ARP" methods in order to make the object "appear" on that subnet, so that other fixed hosts on that subnet are unaware that there is an extra "hop" required to get to that object.

For mobile objects, it may prove necessary to use Mobile IP [7] to provide connectivity for an object on a foreign Networked Surfaces; in this case, the surface manager would be an ideal place to execute the "foreign agent" component of Mobile IP.

**3.2.4. B-LVDS Link Layer.** As discussed, in the prototype system, high bandwidth devices (i.e. IP-capable devices) will use busses employing the B-LVDS modulation system. The modulation system itself provides no addressing or arbitration functions. This must be done in software, and is performed by the device driver's B-LVDS "engine".

The packet format used is very simple - a byte for the dynamic address listed above, a byte for a packet "type", and then a variable number of data bytes. There is no error detection, as this is expected to be done at higher layers in the protocol stack.

For arbitration, various forms of implicit grant systems, such as carrier sensing and collision detection (CSMA/CD) were rejected in favour of an explicit grant system, similar to token bus. This is because there is an obvious candidate for a centralised bus manager on Networked Surface busses, namely the surface manager, which is aware of all the participants on the bus (a highly dynamic quantity), and can therefore appropriately assign bandwidth.

This model comprises an explicit grant message from the surface manager to each object in turn, to which the object either replies with a data packet, or with a "grant reply" message indicating no data. This also allows the surface manager to track disconnection of objects, using a timeout for the object to either send data or a grant reply. Since the model involves passing of the token back to the surface manager between each object, the term "token star" is used to describe the arbitration system.

## 4. Measurements

Though the current prototype is still undergoing development, preliminary measurements have been carried out to validate the practicability of the proposed design.

This section presents initial measurements in two areas. Firstly, the time taken for an object to establish a connection with the surface is measured. Secondly, the bandwidth available on the B-LVDS data bus is explored.

### 4.1. Connection Measurements

As explained earlier, connection takes place in two distinct phases. The first phase involves execution of the handshaking protocol between the tile and object controllers. The second phase involves registration of the object with

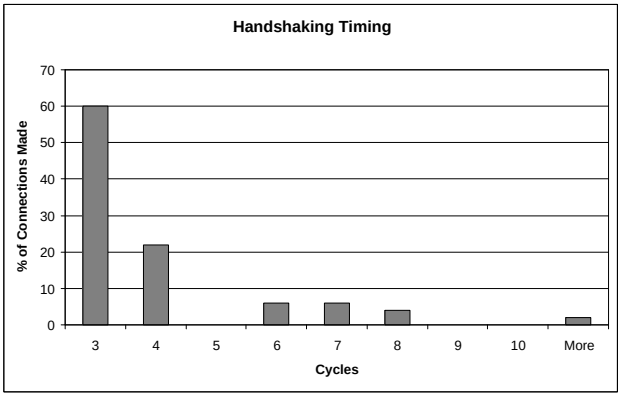


**Figure 8. Handshaking Timing Measurements (1 cycle = 112.5ms)**

the surface manager. Handshaking is expected to take by far the larger share of time.

In order to make an estimate of the expected handshaking durations, it is noted that the tiles operate in "cycles,". Due to the handshaking protocol details presented earlier, an error-free handshake is predicted to take three cycles.

The cycle time is set at 112.5ms, this being the time for one tile to spend approximately 4.7ms for each of its 24 strips. The time per strip is determined by the processing and communication time demands of the prototype hardware.

In order to test this behaviour, an object was placed at various random positions on a tile. Ideally, the object would be placed on the surface and a timer started simultaneously, however this is difficult to achieve without the use of a means of detecting the placement (such as pressure sensors).

The experiment therefore involved placing an object on the surface, but disabling handshaking until a button was pushed. The use of a button allowed a timer to be started simultaneously with handshaking.

Two times were measured for each connection: the time from the button press to a completed handshake, and the time from a completed handshake until registration was complete. Ten measurements were taken for each of five random object placements on the surface, giving a total sample size of fifty.

Figure 8 shows a histogram of the handshaking times, given in terms of tile cycles. The times of the registration phase is not shown, as it was consistently measured below 7ms, and therefore plays little part in determining the connection time.

As expected most connections take place within three cycles. Some require one additional cycle, possibly due to bit errors during handshaking.

There is also another group of connections which is established in around 7 cycles. There is no obvious explanation for this group; further investigation is necessary.

The key result of these measurements is that in over 80% of cases the object was connected in under half a second, which is not a significant delay as perceived by humans.

### 4.2. B-LVDS Data Rate Measurements

Tests were also done in order to obtain preliminary measurements of the available bandwidth of a B-LVDS data channel.

For various data rates from 1 Mbit/s to 2.8 Mbit/s, ping packets of different sizes were sent between the manager and an LVDS object. 100 pings were sent for each experiment, and the average ping time was recorded. The results are shown in Table 1.

| Bit Rate (Mbit/s) | Packet Size (bytes) | % Packet Loss | Average Ping Time (ms) |
|---|---|---|---|
| 1 | 80 | 0 | 1.9 |
| 1 | 530 | 0 | 10.1 |
| 1 | 2030 | 0 | 38.4 |
| 2 | 80 | 0 | 1.2 |
| 2 | 530 | 100 | n/a |
| 2.8 | 80 | 0 | 1.0 |
| 2.8 | 530 | 100 | n/a |

**Table 1. Bandwidth Experiment Results**

The cases without packet loss show that the data channel and prototype hardware cope adequately with these data rates. However, there are experimental failures for 530 byte packets at 2 Mbit/s and 2.8 Mbit/s, which can be explained with a closer look at the current object prototype.

The object consists of a laptop using a PCMCIA digital I/O card to create a bidirectional interface to the object pad controller. This introduces a bottleneck, in that the PCMCIA interface used has been found to be unable to keep up with LVDS bus speeds higher than 1Mbit/s. This was confirmed by examining the log files for the PCMCIA driver, which shows "FIFO empty" errors from the object hardware during the lossy experiments.

However, small packets were transmitted without any problem, because the LVDS object hardware includes a FIFO to buffer packets, which is currently 127 bytes. This allowed the small packets to be written completely into the hardware at the start of each transmission.

Higher bandwidths were not explored because of another bottleneck, in that the object controller hardware does not include an oscillator fast enough to receive data at speeds

over 2.8 Mbit/s. Both of these bottlenecks are the subject of further improvements.

In summary, these results show that the LVDS data channel can support bandwidths of at least 2.8 Mbit/s without introducing significant bit errors (i.e. there were no bit errors in 8000 bytes of data sent at 2.8Mbit/s during this experiment). The results also confirm that prototype hardware is currently subject to bottlenecks preventing general usage at rates over 1 Mbit/s.

## 5. Conclusions

Networked Surfaces provide mobile networking and power without the use of wiring or radio. They offer the user an intuitive and consistent way of connecting a broad range of devices, while simultaneously providing location information about the connected devices.

Devices connect to the Networked Surface by negotiating with autonomous tiles using a handshaking protocol, and subsequently registering their presence with a Surface Manager. The Surface Manager acts as bridge from the devices to the internet, arbitrating access to the high speed data busses using the "token star" scheme presented. The Surface Manager is implemented as a Linux PC containing a custom PCI card, a kernel device driver and a user-level daemon.

This implementation offers a flexible test bed for continued research into issues from physical layer modulation to transport layer handling of frequent disconnections.

## 6. Acknowledgements

The authors would like to thank everyone at the LCE and many people at AT&T Laboratories Cambridge for helpful discussions and comments.

We would like to thank Ant Rowstron, formerly of the LCE, for his valuable ideas in the initial stage of this project.

Frank Hoffmann is pursuing a PhD, funded by AT&T Laboratories Cambridge.

James Scott is also pursuing a PhD, funded by the Schiff Foundation of Cambridge, and by AT&T Laboratories Cambridge.

Mike Addlesee and Glenford Mapp are research engineers at AT&T Laboratories Cambridge, where Andy Hopper is Managing Director. Andy is also Professor of Communications Engineering at the Engineering Department of the University of Cambridge.

## References

[1] Bluetooth. A low-cost short-range radio networking solution. http://www.bluetooth.com/.

[2] BLVDS. A high speed multi-drop bus signalling architecture. http://www.national.com/appinfo/lvds/.

[3] A. Hopper. The Clifford Paterson Lecture, 1999. Sentient Computing. *Philosophical Transactions of The Royal Society of London*, 358(1773):2349–2358, August 2000.

[4] $I^2C$. A networking solution for integrated circuits. http://www-us2.semiconductors.philips.com/i2c/.

[5] IEEE. *ANSI/IEEE Std 802.11*. IEEE, December 1999.

[6] Intel. *i960 Hx Microprocessor Developer's Manual*, chapter 15. Intel, 1998.

[7] C. E. Perkins. Mobile networking in the internet. *ACM Mobile Networks and Applications*, 3:319–334, 1998.

[8] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 85–90, Santa Cruz, California, USA, Dec. 1994.

[9] J. Scott, F. Hoffmann, M. Addlesee, G. Mapp, and A. Hopper. Networked Surfaces : A New Concept in Mobile Networking. In *Proceedings of Third IEEE Workshop on Mobile Computing Systems and Applications*. IEEE, December 2000.

[10] T. G. Zimmerman. Personal Area Networks: Near-field intrabody communication. *IBM Systems Journal*, 35(3,4):609–617, 1996.