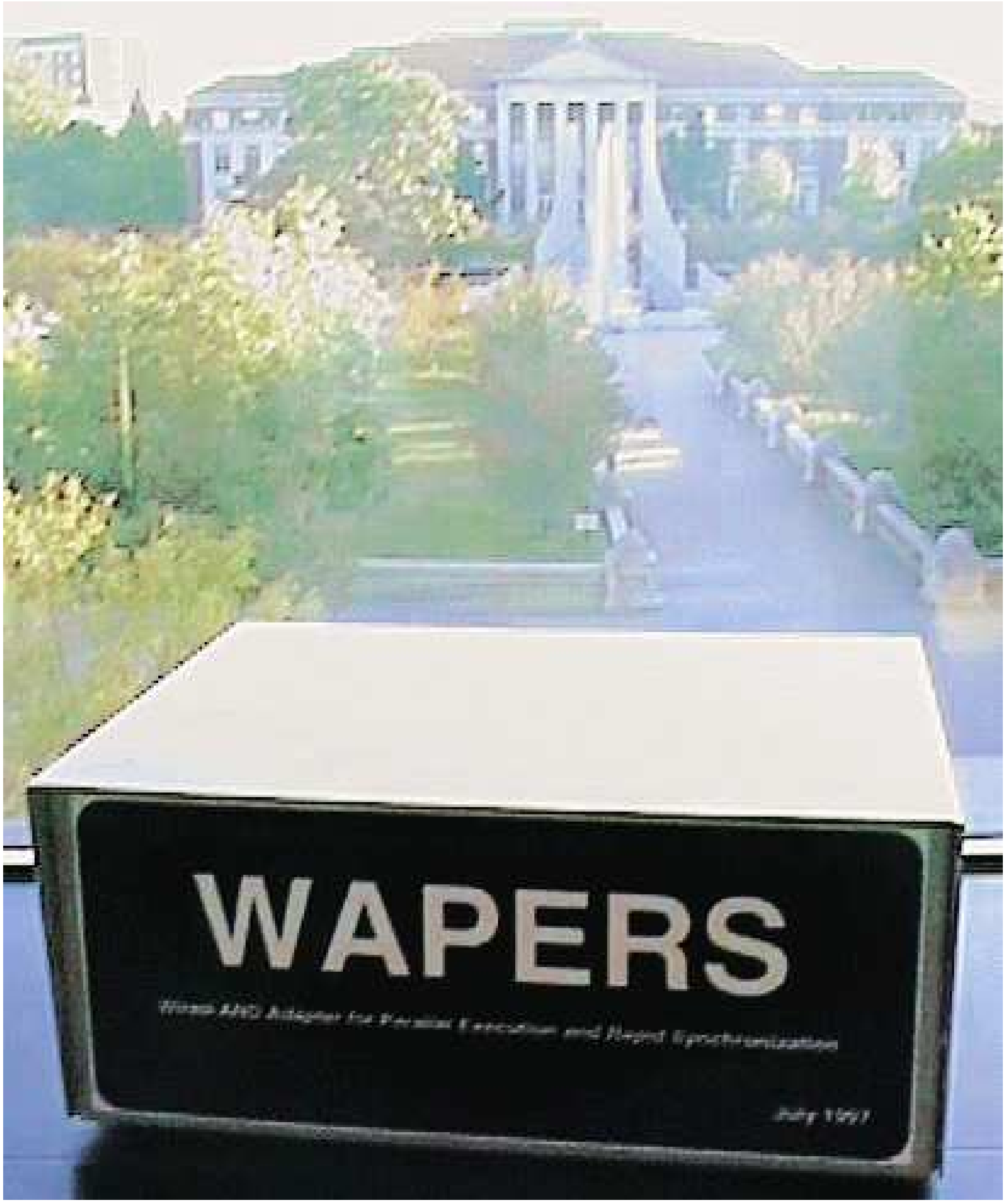
WAPERS: The Wired-AND Adapter for Parallel Execution and Rapid Synchronization



WAPERS, February 10, 1999

Hank Dietz

School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285
hankd@ecn.purdue.edu

Abstract

PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization) has proven that a group of ordinary PCs and/or workstations can function as a tightly-coupled parallel system. Unfortunately, PAPERS is hardware. True, TTL_PAPERS is *very simple* hardware, but it is still a separate box, with its own power supply, that you build or purchase.

In contrast, although WAPERS supports the full user-level AFAPI (Aggregate Function Application Program Interface), the WAPERS hardware is entirely passive. Literally, WAPERS is a wiring pattern. How does a wiring pattern implement aggregate functions? The basic building block for TTL_PAPERS aggregates is NAND; WAPERS replaces this by a wired-AND of open-collector outputs from standard parallel ports (SPPs). Depending on the precise electrical properties of the SPPs and cables, WAPERS is even modularly scalable.

What's the catch? There are three. (1) WAPERS yields somewhat lower performance than TTL_PAPERS. (2) Scaling may be limited to as few as about 8 machines. (3) **WAPERS can fry parallel port hardware if things are not configured correctly.** However, for a lot of small-scale cluster applications, WAPERS is an amazingly simple way to get the performance you need.

1. Theory of Operation

Although the concept of using the SPP for communication between machines is widely accepted (e.g., using "LapLink" cables), few would think of it as a viable approach for parallel processing... but why not? The traditional views of parallel and distributed processing rest on a set of basic assumptions that are incompatible with achieving good performance using such an interface. So, WAPERS does things differently:

- Conventional wisdom suggests that the operating system should manage synchronization and communication, but even a simple context switch to an interrupt handler takes more time than WAPERS takes to complete a typical synchronization or simple communication. All interactions with WAPERS are I/O port accesses made directly from the user program; there are no OS modifications required and no OS call overhead is incurred.
- Communication operations are characterized primarily by latency (total time to transmit one object) and bandwidth (the maximum number of objects transmitted per unit time). The hardware and software complexity of most interaction methods results in high latency; high latency makes high bandwidth and large parallel grain size necessary. In contrast, WAPERS is very simple and yields a correspondingly low latency. Providing low latency allows WAPERS to work well with relatively fine-grain parallelism, but it also means that relatively low bandwidth can suffice... which is

good, because WAPERS normally uses a one bit wide data path that provides *very limited bandwidth*.

- A typical parallel computer is constructed by giving each processor a method of independently performing synchronization and communication operations with other processors; in contrast, WAPERS interactions between processors are performed as aggregate operations based on the global state of the parallel computation, much as in a SIMD machine. This SIMD-like model for interactions results in much simpler hardware and a substantial reduction in software overhead for most parallel programs (as was observed in the PASM prototype). For example, message headers and dynamically-allocated message buffers are not needed for typical WAPERS communications. It is also remarkably cheap for the WAPERS hardware to test global state conditions such as **`any`** or **`all`**.

Thus, WAPERS does not perform any magic; it merely uses a parallel computation model that naturally yields simpler hardware and lower latency... and even though it is an electrically passive design, WAPERS does implement the most important functions directly in hardware. The low-latency synchronization and communication WAPERS provides allow users to take full advantage of the "loosely synchronous" execution models associated with fine-grain to moderate-grain parallelism.

1.1. WAPERS Vs. PAPERS

For most versions of PAPERS, especially TTL_PAPERS, bitwise **NAND** plays a key role in implementing both barrier synchronization and other aggregate functions — but WAPERS does not contain **NAND** gates. The TTL_PAPERS barrier logic also depends on a single bit of state stored in a flip-flop, which WAPERS lacks. In fact, WAPERS contains no active logic at all.

An SPP actually has three separate I/O registers, which are mapped to consecutive I/O addresses starting at 0x278, 0x378, or 0x3bc. The first register is an 8-bit data output port, which WAPERS could use to implement an 8-bit broadcast bus, but that we generally ignore for electrical reasons. The second register is a 5-bit status input port that serves no purpose for WAPERS. The last register is a 4-bit control output port, which WAPERS uses to implement the 4-bit **AND**.

1.2. Wired-AND

Each bit of the SPP control output was originally an open-collector TTL signal pulled-up to +5 volts through a 4.7K ohm resistor, and the logic level of each signal can be read back by reading the feedback register at the same I/O address. Three of the four control outputs are inverted; actually, all four bits were originally driven by 7405 inverting open-collector buffers, but the 0x04 bit was inverted twice on output while the other three bits were inverted once on output and once on input via the feedback register. Although the different paths can yield slightly different characteristics, we can essentially ignore the differences except in the lowest-level WAPERS software, which must correct for the fact that three lines have the inverted sense.

The open-collector outputs are only driven low; they slowly drift high through the pull-up resistance. Normally, reading the value on the feedback port will get you what you output — however, when a bit is set to the high output value, an external connection could harmlessly pull the signal low by sinking the current provided by the pull-up resistance. Thus, each of these control lines is simultaneously both an output and an input. If we connect several of these I/O lines, the voltage is logic high only if all the outputs were high, i.e., all the signals will be **AND**ed together.

Does this wired-**AND** *really* work? Well, yes, but there are a few constraints:

- Assuming a 4.7K ohm resistor is present in the port hardware as a pull-up for each line, pulling *n* connected lines to low logic level requires sinking approximately *n* miliamps (˜1ma per 4.7K resistor to +5 volts). In the worst case, a single driver must sink the current from all *n* pull-ups. Fortunately, TTL is relatively good at sinking current; some TTL parts can sink over 20ma... which would allow quite a few machines to be connected. Unfortunately, the 7405 may only be able to sink about 8ma, limiting scalability to about 8 machines.
- Although the open-collector TTL implementation described above was the original de-facto standard, as time has passed, it has become very unusual for a parallel port to be implemented with a handful of MSI TTL parts: now, it is typically a single CMOS chip. The result is that modern implementations of SPPs often have very different electrical properties, especially for the open-collector outputs. In theory, one could make a "better" open-collector port using CMOS, but the fact that these ports are commonly implemented as single chips tends to severely limit heat dissipation, and thus limits the sink current. In short, it is likely that CMOS open-collector control output implementations will be roughly compatible with open-collector TTL, but the pull-up and sink abilities might be quite different.
- Ironically, as the number of machines increases, the signal quality and noise immunity typically will get better — until things stop working. This is because a weak pull-up will generally yield a very slow signal rise time; multiple weak pull-ups combine to yield a stronger pull-up and sharper rise time. The signal fall time generally should not be a problem, because even a single open-collector TTL output is quite effective at actively sinking the signal to ground. The catch is that, when *n* gets too large, the system may still work for a while, but the driver sink current specifications may be intermittently exceeded. The expected result would be that, if the same single output is held low sinking too much current for an extended period of time, the driver is likely to build-up heat until a thermal failure of the driver occurs. As WAPERS uses the port, this is most likely to occur on the 0x01 bit of the control port, which serves as the data path (the three other signals change state with every two barrier synchronizations performed). Because you really do not know precisely how much current your port drivers can safely sink, you should avoid using the built-in ports of your computer for connecting WAPERS clusters, especially clusters with more than about four machines: it is better to fry an $8 ISA card than a $200 motherboard.
- Perhaps the most scary aspect of modern parallel port implementations is that there are now at least four "standard types" of parallel ports: SPP, PS/2 (often called bidirectional), EPP, and ECP. The good news is that the more advanced ports are described in a nice, clean, IEEE standard (1284). The bad news is that, although all the advanced ports are supposed to provide the basic functionality of SPP, some ports only do so when configured in an SPP-compatible mode. In the higher-speed modes, it is common for the control output bits to switch from open-collector drivers to regular

TTL or CMOS drivers to support faster data transfers with better noise immunity. Wired-**AND** of regular drivers may work for a while, but is definitely not recommended. The worst case would be a single CMOS driver pulling high against a group of drivers pulling low; the CMOS pulling high may try to source more current than it can dissipate the heat for. The only remedy for this is to check that your ports are configured to support open-collector control output, which may involve setting jumpers or changing the I/O configuration information in the power-on setup menu of each computer. Worst case, we have found one computer in which the motherboard's parallel port control output cannot be configured as open-collector... our remedy for this is one of the aforementioned ISA card ports.

- **Do not use parallel port "pass through" devices on a port that is being used for WAPERS.** Electrically, all bets are off if you use one of these. In particular, these devices may duplicate only a fraction of the properties of the full port, e.g., often they do not implement the open-collector outputs as such. Worse still, these devices often suck power from the port outputs (using diodes and a DC-to-DC converter), further compromising the electrical properties of the signals.
- Be aware that there was a generation of CMOS port hardware that was highly prone to damage from ESD (electro-static discharge). Some of these ports are still around, and these are why most parallel port devices still warn that you should not "hot plug" devices: power should be off on both the device and the port when cable connections are made or broken. In truth, TTL is not very sensitive to this type of problem, and modern CMOS tends to have elaborate internal protection against this type of damage, but.... We mention this because you might think that WAPERS, having no active components, could not cause such problems to surface — in fact, WAPERS is actually more likely to expose this type of problem, because the *n* machines connected to WAPERS are effectively *n* active, independently powered, devices on each parallel port.

Well, maybe it is more than an few constraints.... ;-) The gist of it is that wired-**AND** of the control port signals will work fine if your systems are properly configured, and using cheap ISA SPP cards is a pretty good way to hedge the bet.

So, what good is a 4-bit wired-**AND** anyway? The answer is that these four lowly signals implement both fast barrier synchronization and bitwise aggregate communication functions. Here is how.

1.2.1. Fast Hardware Barriers Without Hardware

Before discussing how WAPERS uses the wired-**AND** signals to implement fast hardware barriers, it is useful to briefly review a bit of the history of PAPERS-style barrier hardware.

A barrier synchronization is an *n*-way synchronization in which each processor:

1. Signals that it has arrived at the barrier.
2. Waits for a signal from the barrier logic indicating that *all* processors have signaled their arrival.
3. Resumes execution after the barrier.

It doesn't take a flash of inspiration to realize that the signal in step 2 is essentially the logical **AND** of signals sent by each of the processors. Way back in 1987 we figured that out, and further realized that by selecting between a constant 1 and the signal out of each processor, the inputs to the **AND** could be controlled to allow any arbitrary set of processors to participate.

The thing we did not realize back then is that, using computers that can receive interrupts from other devices, a second barrier synchronization mechanism is needed to confirm that the barrier logic has been reset by all processors before any processor can attempt another barrier synchronization on the primary unit. The result was that our first barrier logic implemented four-cycle barriers:

1. Output "waiting at barrier" signal
2. Input "all are at the barrier" signal
3. Output "waiting for reset" signal
4. Input "all are reset" signal

We then had the insight that, by using a set/reset flip-flop, we could combine steps 1 and 3, and also steps 2 and 4, to create a two-cycle barrier system. In this scheme, two barrier **AND** (actually **NAND**) trees are used such that when waiting at one, we are resetting the other. This is the two-cycle design used in most PAPERS units, including the widely disseminated November 1994 TTL_PAPERS design.

Without the flip-flop, it is not possible for WAPERS to perform two-cycle barriers using just two **AND** trees. The problem lies in the fact that the previous barrier signal must still be available for other processors to read while some processors are initiating the next barrier. Thus, the previous barrier cannot be reset until after the next barrier has completed — and some processors may be starting a third barrier. The somewhat strange conclusion is that by cycling through three separate barrier **AND** trees, WAPERS can implement two-cycle barriers. At any barrier, WAPERS is essentially preserving the state of the previous barrier signal while checking-in at the next barrier and resetting the third barrier.

Bits 0x08, 0x04, and 0x02 of the control register are used as the three barrier signals. The slight additional complication in this assignment is that the sense of bit 0x04 is not inverted, while the other two bits are inverted. This is compensated for by the lowest-level WAPERS port I/O software.

1.2.2. Bitwise Aggregate Function Communication

Given a 4-bit control output in which three lines are used to implement barrier synchronization, only a single line (corresponding to the 0x01 bit) is available to implement data communication.

The bad news is that transmitting data using a single bit path is essentially software-intensive serial communication, and the raw data rates are not much better than high-speed RS232C serial ports — typically between 100k and 200k baud. However, there are a few important differences that make this much more capable than just a multi-tap serial line:

- Although the bits sent can be broadcast from a single processor (by having all other processors output a logic high level), it is also possible to directly use **AND** of one bit from each processor. For example, this makes it possible to determine if any processor meets some condition (**`p_any(f)`**) using a single communication operation.
- While multi-tap serial lines use collision detection and start and stop bits to acquire the channel and delimit transmissions, there is no need for such mechanisms here. The barrier logic ensures that all processors act in a synchronously orchestrated fashion without any conflicts or ambiguity. Each bit transmission is associated with two barrier synchronizations: one to ensure that all processors have read the previous bit value, and a second to ensure that no processor reads the new value until all have contributed their new bit value. The two barriers take four cycles; a fifth cycle is added to separately sample the data. This makes the transmission reliable by giving more time for the data value to settle, thus avoiding a possible race between the data value and the barrier signal which is sent simultaneously out the same port. It is essentially the same five-cycle transmission logic used by TTL_PAPERS.
- Unlike serial lines fed by parallel-to-serial converters, the WAPERS software does not need to determine a-priori what bit sequence it will send; the next bit to send can be determined as a function of the **AND** data obtained thus far. This is not a minor improvement, but a major mechanism for compressing and optimizing transmissions based on incrementally-updated global state. For example, one would expect an *n*-processor **`p_putGet32(d,s)`** to transmit 32**n* bits. However, by simply using a one-bit **AND** (**`p_any(f)`**) to determine if any processor wants to read each datum before we send it, we can detect which 32-bit data transmissions are not needed, and can skip them. Another example is that WAPERS can obtain the maximum or minimum *k*-bit value from *n* processors in just *k* bit transmissions, even for floating point values, entirely independent of *n*! The WAPERS AFAPI implementation uses such techniques extensively, typically yielding effective bandwidth far greater than the hardware directly provides... for example, broadcast bandwidth often appears to be between 2 and 9 times better than it actually is.

Thus, although one would expect WAPERS to yield about 1/4 the aggregate function performance of TTL_PAPERS (which has a 4-bit data path), actual performance using the 1-bit **AND** is often much closer to that of TTL_PAPERS. In fact, trying to take advantage of the 4-bit pathway of TTL_PAPERS makes it difficult to implement some optimizations, so WAPERS will occasionally outperform TTL_PAPERS.

Notice that here we have only hinted at the complexity of some of the WAPERS AFAPI routines... remember that the full source code is freely available, and it provides the most detailed and up-to-date reference for the algorithms used.

1.3. 8-bit Broadcast Bus

The standard WAPERS AFAPI *requires* the 4-bit **AND** described above, but it is also possible to implement an 8-bit shared broadcast bus without arbitration hardware... if your port hardware can support it.

The 8-bit data output register is accompanied by an 8-bit feedback register than can be used to read the state of the output signals. Originally, the 8-bit output was driven by a 74LS374, which is an octal TTL driver with tri-state outputs... so it is natural to think of placing the 74LS374 in the high-impedance output state and using the feedback register as an 8-bit input register. The catch is that the SPP simply hardwired the tri-state control to always enable output, so the input trick did not work unless you literally cut a board trace and installed a jumper (of course, cutting a trace and installing a jumper within a single-chip CMOS implementation of SPP is a bit difficult ;-). Besides, if you did that, you had an input-only register; to implement an 8-bit broadcast bus, we need software control over the tri-state signal.

Before we discuss the port configurations that do allow us to have an 8-bit bus, it is useful to ponder what will happen if we use cabling that connects these 8-bit SPP outputs anyway. Why? Well, it is nice to have just one cable wiring specification, WAPERS can detect and avoid using the faulty 8-bit bus, and full WAPERS functionality on appropriate ports can actually be implemented by simply connecting the corresponding pins across all the ports, which is particularly easy (discussed in section 2.2 as Design 2). So, what happens when actively driven, supposedly "TTL-compatible," outputs are tied together?

We engineers always have been taught not to tie actively driven outputs together, and port hardware varies quite a lot, so surprisingly few people really have a feel for what will happen if you do this. After surveying *many* of my colleagues, I've formulated the following answer. First, things will probably slowly (compared to microsecond port access times) overheat and die if two drivers on a single line disagree; this is probably a larger problem with CMOS than with TTL. We would expect the worst case to be a single CMOS driver pulling high against a group of drivers pulling low; the CMOS pulling high may try to source more current than it can dissipate the heat for. So, let's assume that all the drivers will be in the same state... which state: logic high or low? Most people figured low was safer, because the drivers all agree on ground (the grounds are connected) and both TTL and CMOS are generally good at sinking whatever differences might exist. In contrast, the logic high may differ quite a bit: from just over 2 volts for TTL drivers to 5 volts (or even sightly higher, depending on power supplies) for CMOS. However, if every line is low and one CMOS driver is set high, that driver will fry quickly, so a couple of people felt all high is safer. The bottom line is that things will only be damaged by thermal problems which take a little while to develop, so having software set all the lines in the same state should be ok, and WAPERS AFAPI versions that use the 8-bit broadcast bus take precautions to ensure that things stay ok (forcing all lines low when in doubt). Basically, if AFAPI outputs a value on the 8-bit bus and sees something else in the feedback register, it assumes that somebody else is not in the high impedance state. Still, software errors could fry hardware, so seriously consider using just a five-wire connection (Design 1) if you have SPP port hardware.

Ok, we now know how to keep an unusable port from frying despite an inappropriate cable, but what ports are usable? At least in theory, the PS/2 (bidirectional) port is ideal, and EPP and ECP can emulate that. The catch is that some ports will only allow software to tri-state disable the 8-bit output in modes where the control register lines are not open-collector. If you have one of those ports, forget about the 8-bit stuff — no version of WAPERS can work without open-collector control lines.

The tri-state control for the 8-bit output is bit 0x20 of the control register. If this bit is on, the 8-bit output is disabled (you cannot disable individual output lines, but only the entire chip). Thus, it is up to the WAPERS AFAPI to ensure that, at any given time, at most one processor has its 8-bit output enabled. How do we do that? The answer is that we use barriers around each 8-bit bus action... not really very different from how we do 1-bit **AND** transmissions. If there is ever any ambiguity about which processor should have access to the 8-bit bus, the ambiguity is resolved using the 1-bit **AND** facility and the resulting conflict-free bus write order is enforced by barriers.

The result is raw bandwidth typically somewhat over 1Mb/s... still slow, but quite usable given that the latency is on the order of a few microseconds. You should notice two things about this bandwidth. First, it is 8x the raw WAPERS **AND** bandwidth and 2x the TTL_PAPERS bandwidth, but, unlike those, implements only broadcast. Second, because it is an 8-bit wide path, it is not as easy to optimize the transmissions, so the optimized WAPERS **AND** bandwidth can actually be better under certain circumstances. In summary, the 8-bit bus is a nice facility, but it is probably best to view it as providing more consistently good bandwidth rather than improving the best case.

As suggested earlier, some versions of the WAPERS AFAPI actually check for availability and proper operation of the 8-bit bus at runtime. This adds a little overhead to some library functions, but is the safest approach that allows the 8-bit bus to be used wherever possible. In general, we do not recommend using 8-bit WAPERS cables unless you are absolutely sure that these connections will be harmless.

1.4. Interrupts

To facilitate some level of asynchronous operation, some versions of PAPERS provide a separate interrupt broadcast facility so that any processor can signal the others. Such an interrupt does not really generate a hardware interrupt on each processor, rather, it sets a flag that each processor can read at an appropriate time. Generally, this facility is used primarily by the parallel meta-OS to enforce gang-scheduling, etc.

Unfortunately, WAPERS does not provide any such mechanism. The user-level AFAPI signals are fully supported, but WAPERS does not provide an “out-of-band” parallel interrupt facility for meta-OS use. We feel this is a minor issue because WAPERS is intended for experimentation with dedicated applications; it is not recommended as the interconnection network for clusters to serve multiple users running/developing multiple parallel applications.

1.5. Purpose

Unlike most research prototype supercomputers, WAPERS is a fully public domain hardware design and software intended to be widely replicated. It is hoped that the fine-grain capabilities of WAPERS, and the various more sophisticated PAPERS units, in linking conventional computers will bring a qualitative change to the fields of cluster, network, and heterogeneous supercomputing.

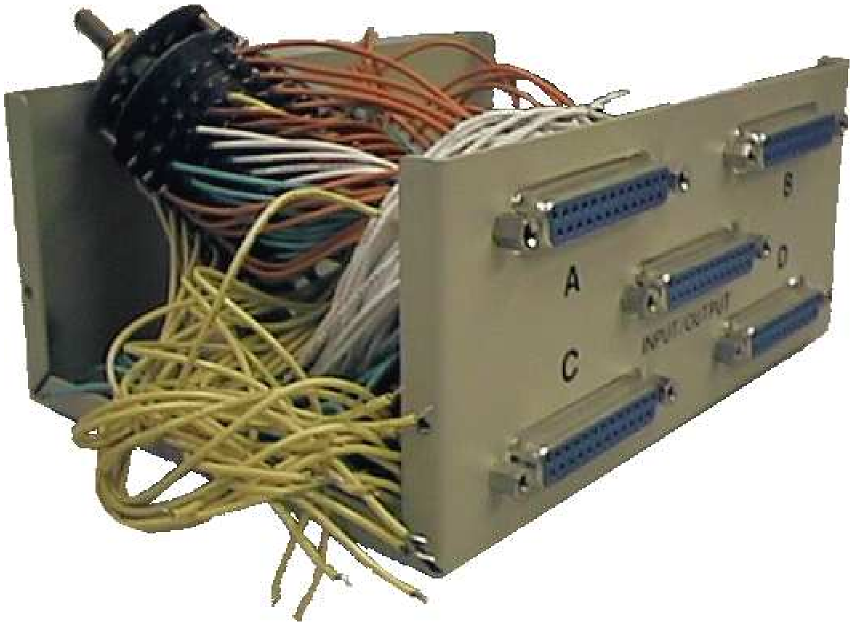
2. WAPERS “Hardware”

As discussed above, WAPERS is not really hardware, but a wiring pattern. In this section, we detail two different ways to implement an appropriate wiring pattern. The first method implements only the bare minimum wiring, but yields a box that can be safely used with any parallel ports implementing the open-collector control outputs. The second method creates a version that is even easier to build and also implements the 8-bit broadcast bus, but doesn't look as nice and is potentially a bit risky to use with ports not supporting tri-state disable of the 8-bit data output.

2.1. Design 1: The WAPERS Box

It is a quirk of our society that it is often cheaper and easier to modify a more complex, but standard, thing than to build something simple from scratch. This design takes full advantage of that fact by modifying a mechanical parallel printer switch to construct a WAPERS unit. A photo of the completed version appears on the cover of this document.

Although you could easily enough buy DB25 connectors, a box, and wires as individual components, for less than $10 you can buy a 4-to-1 mechanical parallel printer switch that contains all of the above, complete with mounting hardware. Do not get a smart all-electronic switch box; you want one that has a dial that you have to turn to select what is connected. When you open the box (probably removing four screws) and look inside, you'll find something like:



Basically, it is 5 DB25 connectors (yielding a 5-machine WAPERS unit) wired to a big, fat, switch. The next few steps may be easier to do if you first remove the switch and connectors from their mountings in the box.

Disconnect the switch by desoldering the wires from the DB25 connectors. A useful suggestion is to always plug a mating DB25 connector to the connector whose pins you will be soldering/desoldering on — this way, even if you get the pins a little too hot, they will not become misaligned when the plastic around them gets soft.

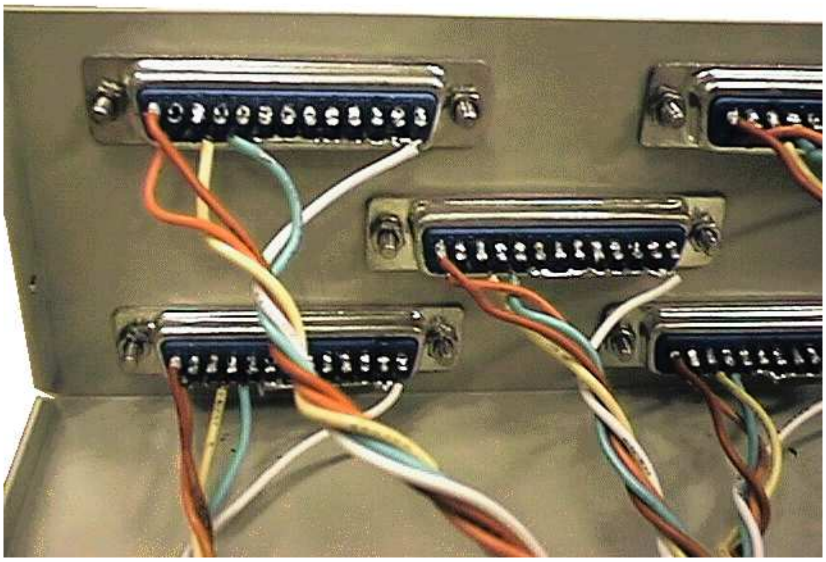
Once you have disconnected all the wires from the DB25 connectors, you have just a few connections to make. The pin/contact assignment for each of the lines is given in Table 1. WAPERS connections are completely symmetric; all PEs are connected identically to five "posts". Table 1 lists the pin numbers in the order they appear on each DB25 connector. Notice that most pins (those not listed) are unconnected.

Table 1: DB25 Pin Connections, Design 1

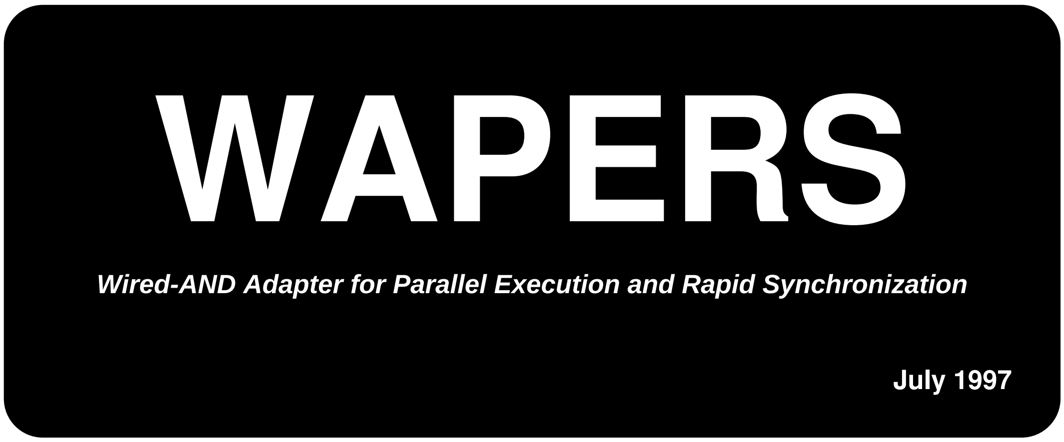
PE Pin #	Post	Standard Name	Use In WAPERS Box
1	Data	Strobe	**AND** Data, 0x01
14	Bar2	AutoFD	Barrier, 0x02
16	Bar4	Init	Barrier, 0x04
17	Bar8	Slct	Barrier, 0x08
18	Gnd	Gnd	Signal ground
19	Gnd	Gnd	Signal ground
20	Gnd	Gnd	Signal ground
21	Gnd	Gnd	Signal ground
22	Gnd	Gnd	Signal ground
23	Gnd	Gnd	Signal ground
24	Gnd	Gnd	Signal ground
25	Gnd	Gnd	Signal ground

How do you make these connections? Odds are that you have a bundle of appropriate-length wires connected to the switch that you just removed, so you can desolder and reuse those wires.

Pins 18 through 25 are all ground and are all next to each other... so start by soldering a wire to pin 25 and then solder-bridging across pins 18 through 25. Then connect a wire to each of pins 1, 14, 16, and 17. Do this for each of the DB25 connectors, and then remount them in the box, twist and solder the corresponding wires together, wrap the soldered-wire connections with electrical tape, and you're done inside. It should look something like this:



The only thing remaining is to finish the box. Closing the box is easy enough (probably replacing four screws), but the front panel probably now has a hole in it where the switch used to be. We recommend covering the front panel, hole and all, with an appropriate label. Here's the one we used:



That's it. You now have a neat little WAPERS unit suitable for connecting up to five machines. Each machine simply gets connected to the WAPERS box via a straight-through DB25-to-DB25 cable.

If your open-collector ports can sink enough current, you can also use this same WAPERS unit as a scalable module. For example, to connect eight machines, you would simply connect four machines to the DB25 connectors of each of two such WAPERS units, and then connect the fifth DB25 connectors of the two units to each other using a straight-through DB25-to-DB25 cable. If you are lucky enough to have parallel port hardware and cables with the right electrical characteristics, you can connect up to 11 machines with 3 units, 14 with 4 units, etc. In general, up to $2 + 3x$ machines could be connected using x WAPERS units, where x>=0. Note that two machines can be connected to each other using a cable without any WAPERS unit... but that cable is essentially Design 2.

2.2. Design 2: The WAPERS Cable

Although Design 1, the WAPERS box, quickly yields a functional and fairly serious-looking unit, there are a few advantages to instead constructing a single, multi-connector, WAPERS cable (Design 2):

1. The box-and-cables packaging of the design may look and feel like a real aggregate function unit, but it also requires a bunch of components: cables and WAPERS unit boxes. Design 2, the WAPERS cable, is literally a single thing no matter how many machines are connected.
2. Although there is soldering and point-to-point wiring involved in making the Design 1 WAPERS unit, the Design 2 WAPERS cable requires neither soldering nor wire routing. It is easier and faster to build.
3. The WAPERS cable naturally includes the ability to use the 8-bit broadcast bus, if your port hardware can support it. Although it could be done, it would take hand routing and soldering eight more wires for the WAPERS box to have this ability.

Of course, the WAPERS cable has some disadvantages too. Perhaps the most important disadvantage is that there is the potential for the 8-bit bus line drivers to fry due to software errors, which cannot happen with the WAPERS box wired as described above. Another disadvantage is that the cable is not modularly scalable; heck, you cannot even change the set-at-cable-assembly-time distance between machines. Also, WAPERS is a custom cable, and thus might be slightly more expensive. Finally, the recommended construction uses unshielded ribbon cable that connects the signal ground lines pin-to-pin rather than grouping them as a single high-quality ground; the expected result is poorer noise immunity.

The wiring pattern for a WAPERS cable is incredibly simple: each pin on every connector is tied to the corresponding pin on every other connector. The resulting signal assignments are:

Table 2: DB25 Wire Use, Design 2

PE Pin #	Standard Name	Use In WAPERS Cable
1	Strobe	AND Data, 0x01
2	D0	Bus Data, 0x01
3	D1	Bus Data, 0x02
4	D2	Bus Data, 0x04
5	D3	Bus Data, 0x08
6	D4	Bus Data, 0x10
7	D5	Bus Data, 0x20
8	D6	Bus Data, 0x40
9	D7	Bus Data, 0x80
10	Ack	Ignored (unused input)
11	Busy	Ignored (unused input)
12	PE	Ignored (unused input)
13	SlctIn	Ignored (unused input)
14	AutoFD	Barrier, 0x02
15	Error	Ignored (unused input)
16	Init	Barrier, 0x04
17	Slct	Barrier, 0x08
18	Gnd	Signal ground
19	Gnd	Signal ground
20	Gnd	Signal ground
21	Gnd	Signal ground
22	Gnd	Signal ground
23	Gnd	Signal ground
24	Gnd	Signal ground
25	Gnd	Signal ground

For the special case of a two-machine WAPERS cable, simply purchase a DB25-to-DB25 straight through cable... that's all you need.

However, to make a WAPERS cable for *n* machines, things are a bit more complex:

1. Determine the minimum necessary cable length between machines as you will have them physically arranged for the cluster. If the machines are literally stacked, this is probably a little less than one foot. Unless the DB25 connectors on the machines are oriented perpendicular to the desired cable path between machines, allow a few extra inches for each so that the DB25 connector can be turned to plug-into the machine. If the sum of all the spacings (total cable length) is much more than ten feet, you should be aware that the ribbon cable will not offer very good noise immunity.
2. Purchase one 25-wire ribbon cable and a DB25 "25-pin D-Subminiature Connector" for each machine. The cable should be as long as the sum of the spacing between the machines; the DB25 connectors should be the type designed to connect to a ribbon cable by simply being pressed into the cable, puncturing the cable to connect to each wire.
3. Arrange the DB25 connectors at the appropriate positions along the ribbon cable, and press them in (working from one end of the cable to the other).
4. Although no termination is needed at the cable ends, you should be careful to ensure that the wires are not exposed in a way that could allow them to short to other wires or the chassis of any of the machines in the cluster. An easy solution is to simply trim the cable close to the last DB25 connector on each end.

The result is a very unobtrusive custom cable that simply chains from the port of each machine to the next. Better still, if you do not want to make this cable yourself, most local computer stores and cable suppliers will make it for you at a reasonable cost.

3. The Soft Side Of WAPERS

When all the network hardware that you have is a bunch of wires, which is all WAPERS provides, it is clearly necessary that a bit of cleverness be employed in the support software. There are actually three major problems that the software must attack: determining the port configuration, avoiding illegal hardware states, and efficiently implementing the AFAPI.

3.1. Determining Port Configuration

The WAPERS software must determine, or at least attempt to confirm the user's specification of, the port hardware configuration being used. TTL_PAPERS and CAPERS use only the minimum port functionality that is common to all types of parallel ports, but WAPERS needs the ports to do more. WAPERS requires open-collector control output, and might additionally use tri-state data output if that ability is present on all machines within a cluster. **It is entirely the user's responsibility to determine the appropriateness of using their port hardware for WAPERS before attempting to use WAPERS, and hardware damage may result from an attempt to use an incorrectly configured port.**

Although we do not know of any test procedure which is 100% safe and effective in determining the port characteristics, the `portinfo` program provided with WAPERS

AFAPI attempts to determine this information. It also attempts to guide you to the highest-performance port and port configuration. Inside WAPERS AFAPI itself, there are only simplified, occasionally executed, checks to confirm that the port configuration matches that specified by the user.

Finding a parallel port consists of looking for something that responds like a parallel port at one of the base I/O addresses where ports are generally found: 0x278, 0x378, and 0x3bc. Typically, one outputs a non-0xff value to the data output port (the base address) and then reads it back — if you get the same value you wrote, it is likely that a port is present. The catch is that this test does not work if the data output is tri-state disabled, so you really want to enable the tri-state output first (turn off bit 0x20 on base + 2).

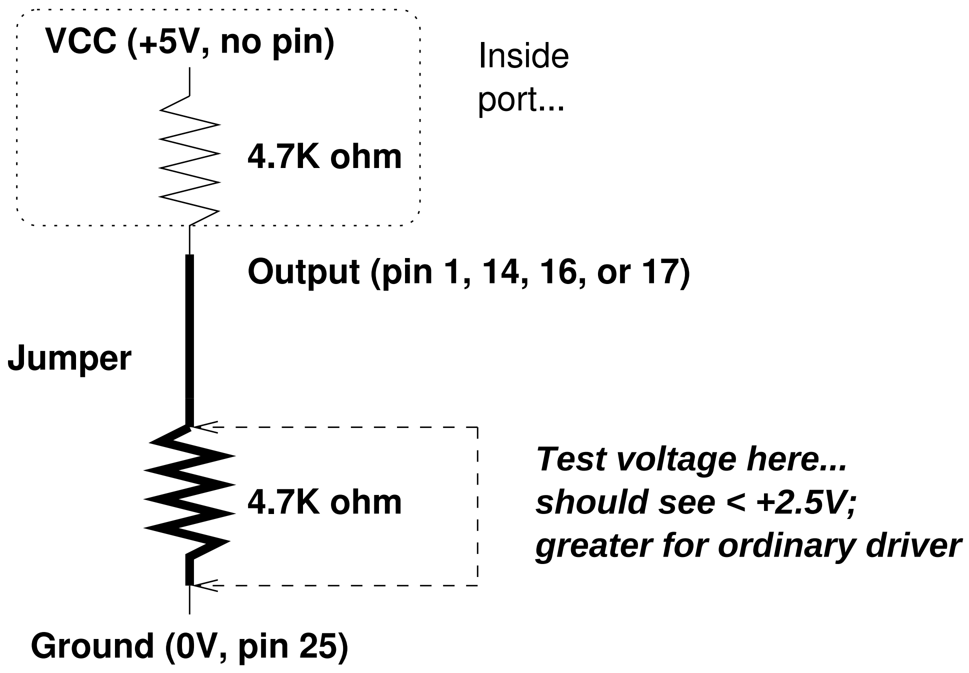
Ok, so it is a port. Is it an ECP port? If so, address base + 0x402 should be the ECP extended control register (ECR). Because 0x3bc + 0x402 yields an address that is generally used by other PC hardware, ports at 0x3bc cannot be used as ECP. If reading the ECR yields a value whose low two bits have the value (value & 0x03 is 0x01), then we have an ECP (whose FIFO is empty and not full).

If it wasn't an ECP, perhaps it is an EPP? The EPP uses the low bit of port address base + 1 as the EPP status; first we must clear that, then write to an EPP extended port address (base + 3, 4, 5, or 6), and then read the value back. The reset generally happens when base + 1 is touched or 0x01 is written there (wouldn't it be nice if such things were truly standardized?). If data output to base + 3 is then seen at base + 3, you have an EPP.

Suppose it was neither ECP nor EPP, is it a simple bidirectional port (also called a PS/2 port)? The way to test this is to tri-state disable the data output port (output 0x20 to base + 2), output some non-0xff data value to base, and then read from base. If what you output is what you read back, your port probably is not tri-state disabled, so it must be an ordinary SPP (Standard Parallel Port).

Ok, hopefully we now know what type of port we have. Ah, but we really do not know enough about it yet — some ports, of whatever type, do not implement the control output lines using the open-collector drivers that WAPERS depends on. So we have to test that the open-collector outputs are indeed implemented as open-collector outputs.

The traditional open collector output configuration in a parallel port uses a 4.7K ohm pull-up to +5V. When the open-collector output is high, the resistance to ground is very large (essentially an open circuit). Thus, if you happen to have a DC volt meter, you can confirm the port construction fairly safely and easily by using the fact that connecting two identical resistors in series between +5V and ground (0V) should yield +2.5V at the point between the resistors. If the port is not open collector, the odds are that the resistance between +5V and the reference point is a lot less than 4.7K ohms, and consequently you will read a value significantly more than +2.5V. The circuit needed to make this measurement is:



No harm should come to the port hardware in either case, since the 4.7K ohm resistance is large enough to keep source current within bounds for an ordinary driver. If your port is not open collector and had an effective resistance of zero ohms to +5V, it would still source only about 1ma (5V divided by the 4.7K ohm resistor you used).

You don't have a voltage meter and/or do not want to deal with connecting a 4.7K ohm resistor? Well, if your parallel port came with a manual, perhaps it is time to do some reading....

In any case, remember: it is **not our fault** if you see a little puff of smoke appear where the driver chips used to be. WAPERS is electrically marginal; we know it, we have warned you, and now you know it. Also, our `portinfo` program is not fully trustworthy. In summary, neither the PAPERS group nor Purdue University can be held responsible for any problems that attempts to use WAPERS may cause.

3.2. Avoiding Illegal Hardware States

The software must ensure that the port hardware settings of all machines always yields a safe global state. This is important because, unlike the TTL_PAPERS interface protocols, if the WAPERS protocols are applied incorrectly, the port hardware of one or more machines can be damaged.

The level of software protection against the occurrence of illegal hardware states varies widely across WAPERS library releases — check the source code for your WAPERS AFAPI version to see how it handles such things. In any case, we are fighting a losing battle in the sense that there always will be some ways in which an undetected hardware error (i.e., noise, short, or other cable problem) or software error could cause port hardware to fry. You can minimize the probability of serious problems by:

- Not using the 8-bit broadcast wiring pattern. The broadcast wiring makes Design 2 inherently more likely to damage port hardware than Design 1.

- Making very sure that your ports support SPP open-collector output and are in an appropriate mode.
- Using cheap add-on parallel port interface cards (EPP/ECP-capable cards will not help performance, and cheap cards minimize your loss if things do fry).

Keep in mind that even the best software protection against illegal hardware states is effective only when that software is running. For example, the port probes done during boot are very likely to cause illegal states at least temporarily, so you might want to phsyically disconnect the machines during boot or to run the WAPERS software to initialize the port within the boot process.

Like the old Saturday Night Live skits about harmless little toys like “Bag O’ Glass” used to say: **Kid, be careful!**

3.3. WAPERS AFAPI Implementation

The whole point of WAPERS is to provide a network supporting low-latency AFAPI communications across two or more machines. Any software that works with the user-level AFAPI will work with WAPERS AFAPI.

Although we now have a unified AFAPI release, the WAPERS AFAPI is customized for the WAPERS “hardware” and what it can do. Thus, WAPERS AFAPI provides the same library interface, but it is not just a minor variation on the TTL_PAPERS AFAPI; it is a re-implementation optimized to yield good performance using WAPERS’ dumb, passive, hardware. To see how the implementations differ, look at the AFAPI sources:

`http://garage.ecn.purdue.edu/~papers/AFAPI/`

4. Conclusion

In this paper, we have presented the complete public domain design of the WAPERS hardware. This design represents the *simplest possible mechanism* to efficiently support barrier synchronization, aggregate communication, and group interrupt capabilities — using unmodified conventional workstations or personal computers as the processing elements of a fine-grain parallel machine.

WAPERS is electrically marginal, and does not scale to very large clusters, but it makes for a very low cost first experience in using tightly-coupled cluster parallel processing. In fact, the ultra-low cost can make WAPERS very appropriate for linking together those wimpy old 386 machines that were in dead storage taking-up valuable office or lab space; even a 386 WAPERS cluster can implement a decent video wall.

If you like WAPERS, but want something better and are willing to deal with more complex hardware, take a look at the various PAPERS designs at:

`http://garage.ecn.purdue.edu/~papers/`

Most importantly, let us know what you think about WAPERS and how you use it. Also tell us if you tried and failed. Send comments to **`hankd@ecn.purdue.edu`**