Simple Serial Protocol (SSP)
Version 2.1

Henry Spencer

henry@spsystems.net

A permanent web site for this specification and related material, and a mailing list for SSP users, are planned but not yet active.

Substantive changes to this document since rev 11 (aka rev 2.1a): document rev numbering changed to reflect protocol version; discussion of version numbering added as section 1.1; section 3.1 added to define and discuss the CRC calculation much more precisely (no change in algorithm, just in description); clarify that master need not fetch entire identity string; more explanation about the intent of the identity string; reinstated the old Appendix A with updated CRC code and pointer to online copies; add note concerning radio use.

1. Introduction

SSP (Simple Serial Protocol) is a simple protocol intended for master-slave communication on single-master serial buses, especially within small spacecraft. Its goals are:

Simplicity. A useful implementation should be possible on a wide variety of hardware, including small microcontrollers with very limited resources.

Efficiency. Bulk data transfers should use most of the line data rate for client data, not overhead (unlike CAN, which in the *best* case is about 60% overhead bits).

No special hardware. Implementation should be possible using standard UARTs and RS-422 transceivers.

Capable. Simple tasks should require little or no extra design work.

Extensible. Customized facilities should fit in easily, without requiring (e.g.) an extra decoding step.

Note that usability over long, loss-prone paths—e.g., radio links—was *not* a goal, and experience indicates that SSP is not ideal for them: its strict request-response structure and lack of a packet serial number make high throughput difficult to achieve over such links. (It has been done but it's messy.)

1.1. Versions

This document specifies Simple Serial Protocol (SSP) version 2.1, which is an upward-compatible extension of version 2 (aka 2.0). See section 11 for a summary of SSP history and changes.

SSP's major version number, before the decimal point, changes only when incompatible changes are made. We don't currently plan any.

SSP's minor version number, after the decimal point, increments when compatible additions are made. The last time that happened was in May 2001; see section 11.2.

This document's revision number is the protocol version plus a suffix letter that indicates a document revision. Document revisions happen whenever it seems useful, usually to improve explanations.

1.2. Terminology

A value or field which is said to be ‘‘reserved’’ is being held in reserve for future extensions to the SSP specification, and is off limits to conforming implementations.

Areas where conforming implementations may vary are described as ‘‘implementation-defined’’, ‘‘application-specific’’, etc.

2. Transmission and Bus Handling

2.1. Physical Layer and Transmission Speed

SSP is nominally intended for implementation on a multidrop bidirectional bus using RS-422 two-wire differential signalling. (RS-485 is an upward-compatible extension of RS-422; its tolerance for multiple simultaneous transmissions is not used by SSP.) However, it is not strongly tied to this. A four-wire RS-422 bus (two unidirectional paths) is also suitable, as is most any other physical layer which does not limit packet length severely.

The normal data format is asynchronous, with one start bit, eight data bits, and one stop bit, at a speed specific to each bus. Data bits are sent LSB (least significant bit) first, as most UARTs do. In the absence of specific reasons to choose another speed, 115.2 kbit/s is the recommended speed: having too much speed is usually better than not having enough, and development is simplified by using a standard PC serial-port speed. (On MOST as of May 2001, the HKC buses run at 9.6 kbit/s and the ACS buses at 115.2 kbit/s.)

The characters of a packet should be sent back to back, with no delays between them if at all possible. Slave nodes should simply wait for more characters if an incoming packet appears to stop in the middle. The master node on a bus must implement a timeout on expected responses (see section 2.3).

2.2. Framing

SSP uses SLIP (RFC 1055) framing, also known in the amateur-radio community as KISS, to break up a stream of characters into frames (each holding a packet). Each frame begins with, and ends with, a `FEND` character (`0xc0`). If `FEND` ever appears in the packet, it is sent within the frame as `FESC TFEND` (`0xdb 0xdc`). If `FESC` ever appears in the packet, it is sent within the frame as `FESC TFESC` (`0xdb 0xdd`). `FESC` followed by any character except `TFEND` or `TFESC` is an error. `TFEND` and `TFESC` are ordinary characters when not preceded by `FESC`.

Each frame contains an SSP packet, as described in section 3. (The higher-level facilities of the KISS TNC protocol, e.g. the first byte being a type code, are not used.) It is an error for a frame to be smaller than the smallest SSP packet.

As discussed in section 3, there is no particular limit on the length of a frame.

The normal algorithm for receiving a frame is simply to collect the bytes of a frame until the terminating `FEND` is seen, and silently discard any empty frames. The presence of a `FEND` at the start of a frame is basically just for robustness, since it flushes out any line noise, or trash left over from an incompletely-received previous frame. Trying to build a more complex state machine, which classifies `FEND`s as start or end, is generally a mistake.

2.3. Transactions and Bus Access

A process on the bus master sends a request packet to a process on some other device, and that process replies with exactly one response packet. This interchange is a *transaction*, and is the closest SSP comes to having a Transport layer.

SSP has no provision (beyond what non-RS-422 physical layers might support) for bus arbitration among multiple masters. Application-specific arrangements must be made to coordinate masters so that only one is master at any given time.

Similarly, unless a non-RS-422 physical layer provides bus arbitration, permission to use a shared bus is implicitly granted to a non-master device only when the master is waiting for a response from it. This means that there can be only one request outstanding at any given time on any particular bus, so a process which cannot complete the request immediately should respond at once with a ‘‘cannot complete

immediately'' response, and provide some way to check on later progress, rather than stalling until it is able to complete the request.

There is still a question of bus hand-over delays—between request and response, and between response and the next request—on buses which use a single data path for both requests and responses. Transmitters on such a bus should get off the bus within one bit time of the end of the last character of a packet; this will typically require hardware assistance or dedicated single-task software. A transmitter waiting to use the bus should wait at least two bit times from the end of the last character of the previous packet; normal processing delays often will suffice to ensure this. (The numbers here are rather tight, but they seem to be realistic with hardware assistance, and longer numbers waste bus time and are a headache to implement. RS232→RS485 converters for PC use must be selected carefully.)

Bus hand-overs between slaves occur even when requests and responses use different paths, since the response path is still (at least potentially) shared. In such situations, slave transmitters should get off the bus within five character times of finishing sending the last character of a response (note that an SSP packet, e.g. the next request, is a minimum of seven characters).

Lack of a response for 250 milliseconds signifies that the destination process did not receive the packet. When the destination is known to respond more quickly, shorter deadlines may be set; if a very long response is expected on a slow bus, it may be necessary to set a longer deadline. Processes are encouraged to respond as quickly as possible, to keep bus delays down. The maximum response delay for each process should be documented.

2.4. Forwarding

When one process is forwarding packets to another, the responsibility for responses lies with the ultimate destination; forwarders just pass (valid) packets back and forth.

Forwarders should be careful about response timeouts: when the source of a request and its destination are separated by one or more forwarders, and the destination does not respond to the request (or the response gets lost en route), the response timeouts in the various bus masters won't necessarily occur exactly simultaneously. In particular, the source might time out before a forwarder does. For this reason (and others), forwarders should make no attempt to report a response timeout: let the source itself time out.

If at all possible, source and forwarder timeouts should be coordinated so the forwarder always times out first. Failing that, forwarders should be prepared to receive a new request while still waiting for the response to an old one, and should consider such a request to constitute a timeout of the old response. (In the extreme case, a forwarder might not know about timeouts at all, just passing packets back and forth and assuming that the source will be smart enough not to send at a time when a reply is expected. This obviously requires careful coordination if more than one source is involved.)

3. Packet Format

The basic format of an SSP packet, before framing is added or after it is removed, is:

dest srce type ...data... crc0 crc1

(This is essentially a scaled-down version of the Ethernet format.)

dest is a one-byte destination address, identifying the *process* which is the destination of the packet. (There might be more than one process in a given hardware device.) A *dest* value of `0` is reserved for possible future use as a broadcast address. Values of *dest* equal to the SLIP `FEND` or `FESC` characters are forbidden to simplify on-the-fly address decoding.

srce is a one-byte source address, identifying the process sending the packet, for purposes of responses etc. A *srce* value of `0` is reserved for future use to indicate an enhanced packet format; ignore packets with that value. Values of *srce* equal to the SLIP `FEND` or `FESC` characters are forbidden to simplify address decoding; ignore packets with those values.

type is a byte containing two fields:

ss.pktype

The bottom six bits are a packet type field, and the top two are supplementary data which may have meaning for specific packet types. (If a particular type makes no use of the *ss* bits, they should be `0`.) Some *pktype* values are predefined to have standard meanings; see section 5. The rest, apart from some reserved values, may have process-specific meanings. As a simplifying notation in the rest of this document, a *type* byte with a particular pair of *pktype* and *ss* values is written *pktype*/*ss*.

data is zero or more bytes of data, the details depending on the packet type and the application.

crc0 and *crc1* are a 16-bit CRC (see section 3.1 for details of the algorithm), sent least-significant byte first, covering all previous bytes in the packet, from *dest* through the last *data* byte. (Note that the CRC is calculated before framing is added or after it is removed.) Ignore any packet with an incorrect CRC. While the CRC is moderately good error protection, it is not perfect; critical operations should be protected by higher-level protocols, to ensure that a single erroneous packet does not cause disaster.

There is no packet-size count in an SSP2 packet's header. The packet length is implicit, since the framing unambiguously delimits the packet. A slave implementation may set an implementation-specific limit on packet size, and ignore all request packets larger than that; the relevant master is expected to know about this (see also section 9) and respect the limit. Slave implementations must not assume that masters can handle response packets of unlimited size, either; when exact response size is left to the slave's discretion, either there should be a documented upper bound, or the request should indicate a maximum acceptable response size. Processes which do not have severe memory constraints are encouraged to be prepared to handle packets of at least 100 bytes.

Aside from limits set by particular implementations, there is no specific limit on the size of an SSP packet. Issues of bus hogging and growing potential for multiple transmission errors (which may not be caught by SSP's simple CRC) do justify some restraint. Packets of circa 1000 bytes bring per-transaction bus-bytes overhead down to circa 2%, so packets much larger than that should seldom be required.

Implementations must ignore any packet which is not addressed to them (or to another process which they are forwarding packets to), and any *invalid* packet: a packet which has a *srce* of `0`, is shorter than the shortest SSP packet, has a wrong CRC, has a framing error or overrun error detected during its reception, or is coming in the wrong direction (ACK or NAK request, or response which is not ACK or NAK).

Implementations are permitted to ignore packets larger than a process-specific limit. NAKing over-long requests is cleaner than ignoring them, but since they *must not* be NAKed unless their CRC checks out, the implementation of this can become rather awkward.

Implementations are encouraged to listen for requests at all times, if possible. Implementations are permitted to be ''deaf''—to ignore incoming requests—when their internal processing requires this, but this is best avoided except while preparing a response (a time when the master should be quiet anyway).

Otherwise, all requests must get responses, if only an indication that the request was not understood. Fields which are supposed to have specific values should always be checked for them; similarly, when the type of the packet dictates a specific length, this should be checked. If a discrepancy is found, abandon processing of the packet and send a NAK/INCORRECT response (see section 5).

3.1. CRC

There is much confusion about the precise definition of CRC algorithms, much of it revolving around bit order. Here we attempt to spell out exactly how the SSP CRC is calculated. The precise combination used is somewhat of a historical accident.

The CRC polynomial is the 16-bit CCITT polynomial, $x^{16} + x^{12} + x^5 + 1$. The initial shift-register value is all 1s. The output is the contents of the shift register, i.e. there is no final XOR operation on the calculated value.

Each data byte is fed into the shift register LSB (least significant bit) first. This is the usual communications convention, but beware: this is backward from the way programmers typically think of bit order, and many software CRC implementations do it MSB first.

As noted earlier, the computed value is sent least-significant byte first. (Since data bytes, as per section 2.1, are sent LSB first, this puts all the CRC bits in order on the wire, LSB first and MSB last, disregarding start and stop bits and complications due to SLIP framing.)

Note that the CRC is calculated before framing is added or after it is removed.

The obvious way to check the CRC of a received packet is to compute the CRC of the header and data, and then compare the bytes of the computed CRC to those of the received CRC. An alternative, often more convenient, is to compute the CRC over the entire received packet—header, data, *and CRC*—which will yield zero if the received CRC matches the rest of the packet.

For testing, the CRC of the eight ASCII characters ‘‘CCITT-16’’ (the sequence `0x43`, `0x43`, `0x49`, `0x54`, `0x54`, `0x2d`, `0x31`, `0x36`) is `0x2364`, which would be sent as the sequence `0x64`, `0x23`.

If your CRC calculation yields `0xaf5f` instead, it is using an MSB-first algorithm. If you reverse the bits in each 8-bit byte going in, and then reverse the 16-bit CRC value coming out, you should see `0x2364`.

If your CRC calculation yields `0xdc9b`, it is XORing the final value with `0xffff` before output. If it yields `0x50a0`, it’s an MSB-first algorithm with a final XOR. (Such a final XOR is common in CCITT-based CRCs, but SSP does not use it.)

The SSP CRC of the nine ASCII characters ‘‘123456789’’ (a common test case for CRCs) is `0x6f91`; the MSB-first CRC for this case is `0x29b1`.

This CRC will detect any single-bit error, any two single-bit errors, any odd number of single-bit errors, and any single burst of errors up to 15 bits long. Packets with more severe damage have a 1-in-65536 chance of yielding the right CRC, so as noted earlier, critical operations should be protected by higher-level protocols (e.g., separate ‘‘arm’’ and ‘‘fire’’ commands).

The following C program fragment (which is hereby released into the public domain, free of all licensing and copyright) computes the SSP CRC, a bit at a time, on a buffer of length `len` pointed to by `bufp`:

```
#define POLY 0x8408          /* bits reversed for LSB-first */
unsigned short crc = 0xffff;
while (len-- > 0) {
       unsigned char ch = *bufp++;
       for (i = 0; i < 8; i++) {
              crc = (crc >> 1) ^ ( ((ch ^ crc) & 0x01) ? POLY : 0 );
              ch >>= 1;
       }
}
```

For systems which are not severely memory-limited, there are faster table-driven algorithms. See Appendix A for more code examples.

4. Data Formats

Most data formats are as required by particular processes. However, in the interests of consistency, the following conventions should be used unless there is urgent reason to do otherwise. All request and response formats defined in this document use these conventions.

4.1. Byte Order

Multi-byte values are sent least-significant byte first (PDP-11, aka ‘‘Intel’’, byte order), so the 32-bit integer `0x01020304` will be sent as `0x04`, `0x03`, `0x02`, `0x01`.

4.2. Floating Point

It is sometimes useful to have a simple floating-point representation, so that telemetry values can be transmitted without needing error-prone prescaling to fit them into integers.

Floating-point values which do not require especially large range or high precision are sent as four bytes: a three-byte fraction followed by a one-byte exponent. The fraction is a two’s-complement integer; its absolute value is 2^{23} times the floating-point value’s mantissa (which is always between 0.5 inclusive and 1.0 not inclusive, i.e. the floating-point value is normalized), and its sign is that of the floating-point value. The exponent is a two’s-complement 8-bit number, signifying a power of 2 from −128 to +127. A zero value is represented by fraction and exponent both zero.

Caution: the fraction does *not* use the sign-magnitude representation which is usual in floating point.

To convert from floating point to this representation, conceptually do the following:

1. If the original floating-point number is zero (either +0 or −0, in implementations which distinguish), both fraction and exponent are `0`. Otherwise...
2. Take the absolute value of the original floating-point number, and normalize it (so all bits to the left of the binary point are `0` and the first bit to the right of the binary point is `1`).
3. The (signed) exponent of the normalized absolute value (the number of bits it had to be shifted rightward to normalize it) is the exponent of the SSP representation.
4. Take the 23 bits to the right of the binary point, and prepend a single `0` bit on the left, yielding a signed 24-bit integer which is positive.
5. If the original floating-point number was negative, negate this integer using two's-complement negation.
6. The resulting integer is the fraction of the SSP representation.

If the absolute value had more than 23 non-zero mantissa bits, step 4 involves eliminating the extras. Whether this elimination uses rounding or truncation is implementation-defined; if it matters to you, you probably ought to be using a different data format!

If the exponent in step 3 will not fit in a signed 8-bit two's-complement integer, the result is implementation-defined. If signaling an error is awkward, supplying the largest number of the appropriate sign (exponent +127, mantissa `0x7fffff` for positive or `0x800000` for negative) is probably the best approach.

To convert back:

1. Convert the fraction (a signed 24-bit integer) to floating point without altering its numerical value.
2. Multiply it by 2^{-23}, so its absolute value is now in the range [0.5, 1.0).
3. Multiply it by 2^x where x is the exponent.

The sequence `0`, `0`, `0x40`, `0x01` represents the value `1.0`. The sequence `0x0`, `0x0`, `0xa0`, `0x01` represents the value `-1.5`.

5. Packet Types

SSP defines some standard packet types so that common needs can be met the same way in all implementations. It leaves a number of packet types available for custom use, so that application-specific needs can be met without having to define another type field and do another decoding step.

The values of the *pktype* field are as follows. Simple SSP implementations might implement only a subset of these; see the next section.

Value	Name	Description	*ss* Bits
0	PING	test packet	0
1	INIT	initialize	flavor
2	ACK	response	flavor
3	NAK	rejection	cause
4	GET	variable fetch	address space
5	PUT	variable store	address space
6	READ	memory fetch	address space
7	WRITE	memory store	address space
8	ID	identify	phase
9		(reserved)	
10 ... 53		custom requests	as needed
54	ADDRT	add route	0
55	DELRT	delete route	0
56	GETRT	read routing table	0
57 ... 63		(reserved)	

The first ten and last ten types have standard meanings, while the rest are available for process-specific custom requests.

The intent of having two standard areas is that the first will be used for common requests which many implementations are expected to provide, while the second will be used for specialized requests which are expected to be less common. This lets simple implementations, which don't need the second-area requests, have a single small decoding table.

Note that a *type* of 27/3, 28/3, or 29/3 would have the same value as the SLIP FESC, TFEND, or TFESC characters (respectively). This is permitted, but discouraged; the simplest SSP implementations might not implement it. (FEND would be a PING/3, which is deliberately forbidden.)

The standard types are defined as follows:

PING Requests a response, with no other action. This is intended primarily for network testing. An ACK in response might use a non-zero *ss* as an indication of the process's general status. A busy process should respond with NAK/FAILED if it can (simple implementations might not even hear the request while busy).

INIT Initialization request, calling for the process/device to (re)initialize itself. INIT with no data is a request for the process to reinitialize itself completely, to its state on power-up; the *ss* bits should be zero in that case. INIT with data specifies a process restart at a specific address, supplied as four bytes of supplementary data (always four, even if the address is shorter; unnecessary upper bits are 0). There might be further process-specific supplementary data specifying other details of how restart is to be done, and the *ss* bits might also control details such as how the address is interpreted. Process-specific restrictions might apply to the restart address, the *ss* bits, and further data; simple implementations might only support dataless INIT.

INIT should always be ACKed immediately, before it is executed. A dataless ACK/0 is acceptable, but the ACK/0 preferably should include two bytes of supplementary data: an estimate of how long the initialization will take, as an unsigned 16-bit number of milliseconds. The process might not respond to packets during initialization. An estimate of 0 is legitimate and indicates very quick initialization with no unresponsive period. A very brief unresponsive period should be indicated with an estimate of 1. An estimate of 65535 (all-ones) indicates an unresponsive period of 65.535 s *or longer*. The estimate is *only* an estimate, a reasonable guess as to when the node will be responsive again; completion of initialization should be confirmed by other means before relying on it. (ACK/NAK response to a PING might be suitable.)

ACK Acknowledgement response, the request has been successfully performed. May or may not include data in response to the request. Especially if no data bytes are present, may use *ss* as a process-specific data value to convey a small amount of detail.

NAK Negative-acknowledgement response, the request could not be performed. The *ss* value may be used to specify a general cause:

Value	Name	Meaning
0	UNKNOWN	request not understood
1	INCORRECT	invalid details in request
2	FAILED	request was attempted but could not be completed
3	-	(reserved)

There might be more (*supplementary*) data bytes, conveying request-specific or process-specific information.

Do not economize on code by using ACK and NAK as ‘‘yes’’ and ‘‘no’’ responses to a request for data (‘‘is the antenna deployed?’’); NAK is reserved for reporting *failed* requests. Use ACK/1 for ‘‘yes’’ and ACK/0 for ‘‘no’’ instead.

GET Request to read from one or more variables, in an address space given by the *ss* bits. See section 7.

PUT Request to store to one or more variables, in an address space given by the *ss* bits. See section 7.

READ Request to read from memory, in an address space given by the *ss* bits. See section 8.

WRITE Request to store to memory, in an address space given by the *ss* bits. See section 8.

ID Request for the process to identify itself, with the *ss* bits specifying what phase of the identification protocol is being done. See section 9 for details.

ADDRT Request for the process (or the device it runs on) to add a packet route. See section 10 for details.

DELRT Request for the process (or the device it runs on) to delete a packet route. See section 10 for details.

GETRT Request to read the routing table. See section 10 for details.

6. Implementation Constraints

An absolutely bare-minimum implementation must listen for a single address, ignore packets addressed to others, detect and ignore invalid or over-long packets, and implement PING, dataless INIT, ID/0, and dataless ACK/0 and NAKs for response. (To get useful work done, it will probably also have to implement either single-variable address-space-0 GET and PUT, or some custom requests.) It should NAK any valid not-overlong packet, addressed to it, which it does not understand. Implementations with more resources available are urged to implement ID/1, the monitoring variables (variable address space 1), and READ and WRITE of at least single bytes/words.

7. Variables and GET/PUT

For many purposes (notably, but not exclusively, T&C), it is desirable to be able to fetch and store small amounts of data. This is so universal a requirement that it seems worth providing as part of the standard protocol.

7.1. Variables

A value which can be read and/or written remotely is known as an SSP *variable*. Variables can be read-only or read-write. Telemetry values typically would be read-only variables. Control settings typically would be read-write variables. Note that there is no provision for write-only variables; commands are best implemented as custom request types, and controls which can be set but whose setting cannot be read back are best avoided entirely.

For protocol purposes, variables conceptually are 32-bit values. Any particular variable might actually implement fewer than 32 bits; in that case, values read from it and written to it are right-justified in a 32-bit

value, with the extra bits on the left all `0`. (Values read from it are padded on the left with `0` bits; values written to it must have `0` in all the unused bits on the left.) Note that since the 32-bit values are sent least-significant byte first, the padding bits are not necessarily in the first byte(s) of the value.

Variables do not necessarily correspond to actual memory locations. For example, the value of a read-only variable indicating a temperature might be read from an ADC when requested (if this can be done quickly enough), rather than being maintained in memory.

While it may be convenient to represent some slightly-larger values as groups of variables, don't go overboard on this. Bulk data (e.g., a star map), especially bulk data of variable size, is better handled by reading and writing it in large chunks by other means.

The variables of an individual process are selected by address. Variable addresses are 16-bit process-specific numbers. An attempt to GET or PUT to a variable that a particular process does not provide is an error (respond with NAK/INCORRECT).

Variables may exist in more than one address space, indicated by the *ss* field of the GET/PUT request. Address space `0` is entirely for process-specific variables, and simple implementations might not implement any other address spaces. Address space `1` contains standard debugging variables, notably for SSP monitoring; see section 7.4. The other address spaces are reserved for future use. Respond with NAK/INCORRECT if access to an unknown address space is attempted.

7.2. GET

To read the value of one or more variables, a GET request is used. Its *data* bytes are a sequence of two-byte variable addresses. There may be a process-specified limit on how many variables can be read with a single GET; it might be as low as `1`. GET does not have side effects, i.e. it can be repeated any number of times without harm (although, of course, the values of time-dependent variables may change).

A successful response to a variable GET is an ACK/0, with the data being a sequence of four-byte values, in the same order as the addresses of the GET. The number of values must be exactly the same as the number of addresses. Conceptually, all variable values are read simultaneously; often this is inherent, but in particular, if the process involved has a notion of update cycles, all values are read from the same cycle.

Observe that if a multi-variable GET assembles the values into its reply buffer starting from the last and working back toward the first, the reply buffer can share space with the request buffer.

7.3. PUT

To write values to one or more variables, a PUT request is used. Any error, that is, any condition causing a NAK (except as discussed below) leaves all the variables unchanged. PUT does not have side effects, aside from the change to the variable(s); to issue a process-specific command, use a custom request, not a PUT to a magic variable.

A PUT request contains one or more sequences of six bytes each, an address followed by a value, like so:

adr0 adr1 val0 val1 val2 val3

There may be a process-specified limit on how many variables can be written with a single PUT; it might be as low as `1`. Conceptually, all variable values are changed simultaneously; often this is inherent, but in particular, if the process involved has a notion of update cycles, all changes occur in the same cycle.

It is an error to attempt to write a value which is too large for the variable, i.e. the actual variable is shorter than 32 bits but the ''extra'' high-order bits are not all `0`. It is an error to attempt to write to a read-only variable, or (of course) to write to a nonexistent variable.

A successful response to a PUT is a dataless ACK/0.

Implementations are permitted to check the variable addresses and values while doing the writes, so that an invalid address or value in the middle of a multi-variable PUT causes the PUT to be only partially completed. Such a case must be reported with NAK/FAILED; the details of how a failure is handled, and of any supplementary data sent with the NAK, are process-specific. (A suggested convention is to do the checking and writes in the order given in the request, give up immediately when an invalid write is encountered, and include two bytes of supplementary data in the NAK/FAILED, a 16-bit integer reporting how

many values were written successfully.) Senders of multi-variable PUTs are expected to ensure that the contents are valid before sending them, if necessary by asking the destination process about any doubtful cases (using process-specific means).

7.4. Monitoring Variables

Variable address space `1`, if implemented, contains standard variables used for monitoring the SSP implementation (and possibly eventually other things). Its contents, currently all read-write, are as follows. (The type `int` here signifies a signed integer of up to 32 bits, whatever size is most convenient for the process; the values are transmitted in packets using the standard 32-bit GET/PUT format.)

Address	Type	Description
`0`	int	character framing/parity error count
`1`	int	receiver-overrun event count
`2`	int	runt packet count
`3`	int	oversize packet count
`4`	int	bad-CRC count
`5`	int	ownership error count
`6`	int	unknown-format error count
`7`	int	wrong-direction error count
`8`	int	response timeout count (0 in non-masters)

A runt packet is one with a non-zero size smaller than the smallest possible SSP packet (which is 5 bytes, including CRC but not framing); usually they will be the result of line noise. An ownership error is receiving characters you did not send when you are supposed to own the bus. An unknown-format error is a packet with a *srce* of `0`. A wrong-direction error is a master receiving a request or a slave receiving a response.

If any of these counts get big enough to wrap around, the resulting values are undefined. Conscientious monitoring code should use PUT(s) to set them back to zeros before that happens.

8. Memory READ/WRITE

Variables are a useful abstraction, but for many purposes, bulk transfers to and from memory are also useful. READ and WRITE are provided for this purpose. Small SSP implementations, e.g. on microcontrollers, might not support these requests.

Memory conceptually may exist in multiple *address spaces*. The *ss* field of a READ or WRITE specifies the *address space* of the memory to be operated on. The *ss* values are process-specific; for example, the 56303 DSPs used on MOST number their three (!) address spaces:

Value	Name
`1`	P
`2`	X
`3`	Y

(The numbering starting at 1 is a historical accident.)

Processes with only one memory address space should number it `0`.

Memory address spaces might overlap. For example, a process which has flash memory might support a ‘‘RAM’’ address space `0` and a ‘‘flash’’ address space `1` which actually have exactly the same contents, except that a WRITE to address space `0` is an ordinary RAM write and a WRITE to address space `1` is a flash write. (Such a process should reject RAM writes to flash and flash writes to RAM. Also, a production version probably should reject all flash writes unless they are first enabled by an appropriate process-specific means, perhaps by setting a ‘‘flash write enable’’ variable to a suitable value.)

Respond with NAK/INCORRECT if access to an unknown address space is attempted.

8.1. READ

To read data from memory, a READ request is used. It has six data bytes, as follows:

adr0 adr1 adr2 adr3 bct0 bct1

adr0 and the subsequent *adr* bytes are the 32-bit address of the memory area to be read. All four bytes are present even if the address space in question is not that large; all unneeded high-order bits are `0`.

bct0 and *bct1* are an unsigned 16-bit count of the number of bytes to be read from the memory area. There may be a process-specific limit on the size of a READ.

There may be implementation-specific restrictions on alignment and size of memory areas read in this way. The simplest implementations might only support one-byte (or one-word) reads. Implementations on word-addressed machines might require the byte count to be a multiple of the word size (as returned; see next paragraph), and might take the address as a word number rather than a byte number (note that the byte count remains a byte count). Any violation of such a restriction is NAKed, as is any attempt to read nonexistent memory.

A successful response to a READ is an ACK/0, with exactly the requested number of memory bytes as data. Devices with a well-defined byte order in memory send bytes in that order. On word-addressed machines with no official byte order, byte order and padding conventions are implementation-defined (for example, 24-bit words might be sent as 3 bytes each, or might be sent as 32-bit values with the top byte zero). The byte count is how many bytes should actually be sent, including all padding.

A READ with a length of zero bytes is an error; respond with NAK/INCORRECT.

8.1.1. WRITE

To write data to memory, a WRITE request is used. (Some SSP implementations might support WRITE only at boot time or only in prearranged areas of memory.) It has a variable number of data bytes, the first four of which are the address, and the rest of which are the data to be written:

adr0 adr1 adr2 adr3 ...data...

All four address bytes are present even if the address space in question is not that large; all unneeded high-order bits are `0`.

The size of the memory area to be written is determined by the number of data bytes following the address.

There may be implementation-specific restrictions on alignment and size of memory areas written in this way. The simplest implementations might only support one-byte (or one-word) writes. Implementations on word-addressed machines might require the number of bytes to be a multiple of the word size, and might take the address as a word number rather than a byte number (note that the byte count remains a byte count). Any violation of such a restriction is NAKed, as is any attempt to write nonexistent or read-only memory.

For devices with a well-defined byte order in memory, bytes are sent in that order. On word-addressed machines with no official byte order, byte order and padding conventions are implementation-defined (for example, 24-bit words might be sent as 3 bytes each, or might be sent as 32-bit values with the top byte zero).

A successful response to a WRITE is a dataless ACK/0.

Implementations should try to ensure that a memory write is not started unless it can succeed completely. For example, checking for nonexistent or read-only memory should be done before the write is started, rather than byte by byte as it is done, so that it can be completed or rejected as a unit rather than partially completed.

A WRITE with a length of zero bytes is an error; respond with NAK/INCORRECT.

9. Identification

In general, masters are expected to know the capabilities of their slaves, so no great provision has been made for inquiring about various aspects of what a process can do. However, it is sometimes useful to be

able to ask a slave to identify itself, so a master can handle a variety of configurations.

The ID request is used to obtain identification information from slaves. The *ss* field of an ID request identifies the *phase* of the identification protocol being used; simple implementations might support only phase 0. Phase values 0 and 1 are currently defined; 2 and 3 are reserved for future use.

9.1. Phase 0

ID/0 is intended as an absolutely minimal facility for determining a few crucial things about a process, including provisions for a small amount of option and version information even on minimal implementations which cannot support much. Normally it is a preliminary to using ID/1 to obtain more detailed information.

An ID/0 request is dataless. A successful response is an ACK/0 with exactly four data bytes:

flgs bsiz ssiz impl

The *flgs* byte is intended for flags indicating information about the response and the process. It currently contains two fields:

rrrr.iiii

The four-bit *rrrr* field is reserved for future use as part of extensions of the ID response, and should always be `0000`. The four-bit *iiii* field has an implementation-defined meaning and can be used to send a small amount of implementation-specific identity information, e.g. options available; it should be zero if not used.

The *bsiz* byte is an unsigned 8-bit number giving the size of the process's packet buffer, as an indication of how large a packet it can send or receive. This size does *not* include framing overhead, so if framing characters (`TFEND`, `TFESC`, etc.) do take up space in the actual buffer, it will be necessary to report a worst-case number equal to about half the real size. A value of 255 indicates a buffer of 255 bytes or more.

The *ssiz* byte is an unsigned 8-bit number giving the size of the identity string obtained with ID/1 requests. A value of 0 indicates that the process does not support ID/1.

The *impl* byte has an implementation-defined meaning and can be used to send a small amount of implementation-specific identity information, e.g. a revision number. It should be zero if not used.

Do not use *iiii* or *impl* for time-dependent information like device status. Use a GET or a custom request instead. These fields are for fixed identification information.

9.2. Phase 1

An ID/1 request has one data byte, an unsigned *fragment number*.

A process which supports ID/1 knows an *identity string* containing information about the process. The maximum length of an identity string is 255 bytes. This string is divided into fragments, in some fixed and predetermined way that is convenient for that process (perhaps fixed-length fragments of the largest size that will fit into its packet buffer). An implementation with a large buffer might fit the whole string into one fragment. The first fragment is numbered `0`.

The response to an ID/1 contains the string fragment indicated by the request's fragment number. Requesting a nonexistent fragment is an error; respond with NAK/INCORRECT.

Normally, the master would use an ID/0 to determine the length of the identity string, and would then fetch string fragments, starting with fragment `0`, until the complete string (or as much of it as interests the master) can be assembled. Although it is normally advisable for the master to fetch the fragments in order, this is not actually required. Nor is the master required to fetch all the fragments; it might stop after getting the information it needs.

The string is a sequence of lines, each ending with a *newline* character (ASCII `LF`, `0x0a`). They supply progressively more detailed information about the process's identity. For example, a possible string (an

imaginary example not representing any real process) is:

```
dynacon.ca Dynacon Enterprises Ltd.
Microwheel 2B
0.8 N-m-s, 1.3beta, options BCF
code build Tue May  1 19:53:50 EDT 2001
```

The first line identifies the process's manufacturer. At the beginning of the line is the manufacturer's Internet domain name (or one of its Internet domain names), as a unique identifier indicating who assigns the names, numbers, and codes used in the rest of the identity string. Optionally, after the domain name may come white space and further information such as a company name; this is to improve human readability.

The second line identifies the process itself, as a model name/number/code/whatever denoting roughly what its capabilities are. No specific constraints are imposed on the format of this. This (interpreted in the context given by the previous line) is meant to tell a master which software driver it should use to talk to this process.

The third line gives the version identification of the process, denoting (together with the previous line) its exact interface and capabilities. No specific constraints are imposed on the format of this. This is meant to inform the driver of exactly what the process can and can't do, avoiding the delays and extra code complexity involved in having to ask the process about details before proceeding.

Subsequent lines give any further information that may be of interest. No specific constraints are imposed on the format of this. This is meant mostly to hold information of possible interest to humans, e.g. exactly when and how the process was constructed, although it is not restricted to that.

If any of this information cannot be written in ASCII, it should be written using Unicode and the UTF-8 encoding (see RFC 2279).

10. Routing Control

Conceptually, when a process sends a packet, it looks in its *routing table* to determine which *port* the packet should be sent through. Many processes will have only one port, e.g. because they reside on a device which only has one SSP-capable interface. Some will have more than one, either simultaneously active (mostly notably in the case of forwarders, see section 2.4) or requiring a choice of which port to use. One notable case is where a device has multiple interfaces with some shared innards, e.g. multiple bus connections with a switch connecting them to a single UART, and is physically incapable of receiving from more than one interface simultaneously.

As a practical matter, a process's routing table will generally be initialized, by process-specific means, when the process is started. Simple implementations might not permit changes to their routing tables. However, in some cases the routing table does have to be changed after startup. Forwarders may need to be configured before they can safely start forwarding, and shared-innards devices will need to select a bus before communicating.

Conceptually, a routing-table entry consists of:

- a *network*
- a *mask*
- a port number
- some flags
- possibly a lifetime

A routing-table entry applies to a particular packet if the packet's *dest* address satisfies:

(*dest* & *mask*) == *network*

The *network* and *mask* values are 8-bit values, like SSP addresses. The *mask* is required to consist of zero or more `1`s followed by enough `0`s to fill out the byte. Bits in the *network* corresponding to `0`s in the *mask* are required to be `0`. Following Internet practice, this pair of values (a *subnet*) can be written in the format *nnn/b*, where *nnn* is the decimal value of *network* and *b* is the number of `1`s in the mask. For example,

`76/8` is a subnet containing only the address 76, and `0/0` is a subnet containing all possible addresses.

Port numbers are unsigned 8-bit numbers; port numbers 250 and above are reserved for special meanings. Currently the only special meaning assigned is 255, which means ‘‘discard’’.

The lifetime, if present, specifies how long the entry will last before it is deleted automatically. Some implementations may not support lifetimes, or may limit their length or implement them slightly imprecisely (e.g. rounding lifetimes to an integer number of seconds).

The routing table may have limited size; simple implementations might support only a single entry. When there are multiple entries, the table is conceptually searched ‘‘most specific first’’, which can be accomplished by sorting it in reverse order by the number of `1` bits in the masks, so any `/8` subnets come first and any `/0` subnet comes last. The first entry that is found, in this order, controls the fate of the packet. A routing table is not allowed to contain more than one entry for the same subnet.

An implementation which does not support changing its routing table might nevertheless support the GETRT request for reading it. Otherwise, if any of the requests for manipulating routing tables are implemented, all of them must be.

10.1. ADDRT

The data accompanying the ADDRT request is one or more routing-table entries, formatted like so:

netw mask flgs port lif0 lif1 lif2 lif3

The first two bytes of the entry are the subnet. The third is a flags field, which is currently entirely reserved for future use and must therefore be `0`. The fourth is the port number. The remaining four are the lifetime for the entry, measured in milliseconds; `0` means no lifetime (that is, an infinite lifetime).

If there is no existing entry for that subnet, this is a request to add a new one. If there is an existing one, this is a request to update its flags, port number, and lifetime. (An update of a lifetime sets the remaining time to the new amount, regardless of how much or how little of the old lifetime remained.)

If the implementation does not support updating its routing table, respond with NAK/UNKNOWN. If the subnet is invalid, the port number is unknown, the lifetime exceeds implementation-defined limits, or a lifetime is specified to an implementation which does not support lifetimes, respond with NAK/INCORRECT. If a new entry is required and there isn’t room, respond with NAK/FAILED. There might be a limit on how many entries can be included in a single ADDRT; it might be as low as 1.

10.2. DELRT

The data accompanying the DELRT request is one or more routing-table entries, with the same format as for ADDRT. This is a request to delete an entry for that subnet. The flags, port, and lifetime values are ignored.

If the implementation does not support updating its routing table, respond with NAK/UNKNOWN. If no such entry is present in the table, respond with NAK/FAILED. There might be a limit on how many entries can be specified by a single DELRT; it might be as low as 1.

10.3. GETRT

The GETRT request is dataless. The response is the process’s routing table, as a sequence of entries using the ADDRT format.

The order of the entries is process-specific and might not be meaningful. In particular, it does not necessarily reflect the search order. (This is intended to permit sophisticated implementations which don’t use a simple sorted-list data structure and might find it awkward to have to generate one.)

It is process-specific whether the lifetime values shown in the response are the original values or the remaining times.

There is an assumption here that an implementation with a non-trivial routing table also has non-trivial packet buffers, so it can send the entire table as a single packet. This seems reasonable.

10.4. Empty Routing Tables and Port Selection

Deletions, either explicit or due to lifetimes running out, might cause the routing table to become empty. This causes the routing table to be re-initialized to the same state it has on startup.

One special case applies here: the shared-innards case, where a newly-initialized device must pick one interface to be its only SSP communications path. In such a case, repeated re-initializations should cycle through a series of initial states, rather than always repeating the same one.

For example, a process with a single switched UART which can be connected to either of two ports might start out by initializing its routing table to a single entry for subnet `0/0` and port `0` with a 500-millisecond lifetime, do the same but with port `1` on the next re-initialization, and alternate thereafter. The cycle would be broken only when a master uses ADDRT to override this routing-table entry.

It is permissible for such a process to ignore all requests except ADDRT during this ‘‘port selection’’ cycling, although it would be polite for it to also answer PING.

11. Change History Summary

11.1. Changes from SSP Version 1 to SSP Version 2

Based on experience with SSP version 1, a number of changes were made, some fairly fundamental, in the design of SSP version 2. This section is an outline of what was changed and why, in roughly the same order as the main text. This is not a complete inventory of changes; in particular, it omits additions to the protocol which present no compatibility problems.

Packet framing is unchanged, except that the one-character delay between packets is gone. Timing specs have generally been loosened; in particular, there is no longer any requirement for slaves to implement receiver timeouts.

There is no longer a length count in the packet header; packet length is defined by packet framing. This removes any hard limit on packet size, shrinks the worst-case packet header, makes all packet headers the same length, and makes more room in the header. There is no longer a one-bit serial number either; it was of minimal value in a master-slave protocol, and no existing implementation checked it. The type number has been expanded, and there is now a small supplementary-data field in requests as well as responses. The larger type number simplifies several things, and provides room for more custom request types.

The authorization to ignore over-long requests was already present (although somewhat obscure) in the final description of SSP1.

Bus-handover delays are now specified. For two-wire buses where requests and responses share a wire pair, the limits are fairly tight, and hardware assistance in getting transmitters off the bus promptly is effectively mandatory. This was much the simplest approach; otherwise it becomes necessary to have quite large worst-case delays. When hardware assistance simply is not available, the best approach is to use a four-wire bus which has separate request and response pairs.

Variable GET/PUT are now separated from memory READ/WRITE. The old MGMT packet type is gone, although its one really useful form (PING) has become a distinct type. The protocol for bus selection is likewise gone, since we no longer have a requirement for it.

ACK and NAK no longer include the type byte of the request. NAK now includes a simple error code using the two-bit supplementary-data field, and any further details are process-specific.

Variables now have address spaces (variable address space 1 has some SSP monitoring variables in it), and variable addresses are now 16 bits. The notion of write-only variables as a second way of giving commands is gone. GET/PUT can read/write multiple variables in a single request.

11.2. Changes between SSP 2.0 and SSP 2.1

The only generally-important difference between SSP 2 (now retroactively renamed SSP 2.0) and SSP 2.1 is that the simplest form of the new ID request is mandatory in a 2.1 implementation. There have been several other additions to the spec, notably routing manipulation, but they are all optional.

12. Acknowledgements and Availability

Development of this protocol was funded by the MOST spacecraft project (itself funded primarily by the Canadian Space Agency), via the Space Flight Laboratory at the University of Toronto Institute for Aerospace Studies.

Henry Spencer designed SSP versions 1 and 2, with much input from several others on the MOST project, notably Dan Foisy (UTIAS-SFL), Doug Sinclair (Dynacon Enterprises Ltd.), and Ron Wessels (Dynacon). HS also wrote the public-domain C code in section 3.1 and Appendix A.

Appendixes

A. CRC Calculation

Here we supply sample C source for three methods of CRC calculation, and a useful related algorithm. These sources are hereby released into the public domain, free of all licensing and copyright. On-line copies can be found under the web address given on the title page.

Here is the most basic version, the code from section 3.1 tightened up and packaged up as a C function:

```
/*
 - ssp2bitcrc - compute SSP2 CRC a bit at a time, somewhat slow but compact
 * Same parameters, algorithm, return value as the old bitcrc().
 */
unsigned short
ssp2bitcrc(data, length)
unsigned char *data;
int length;
{
#       define POLY   0x8408        /* bits reversed for LSB-first */
        unsigned short crc = 0xffff;
        unsigned char *bufp = data;
        int len;
        int i;

        for (len = length; len > 0; len--) {
                unsigned char ch = *bufp++;
                for (i = 8; i > 0; i--) {
                        crc = (crc >> 1) ^ ( ((ch ^ crc) & 0x01) ? POLY : 0 );
                        ch >>= 1;
                }
        }

        return crc;
}
```

Here is a considerably faster version, using table lookup:


```
/*
 - ssp2bytecrc - compute SSP2 CRC a byte at a time, fast but 512-byte table
 * Same parameters, algorithm, return value as the old bytecrc().
 */
unsigned short
ssp2bytecrc(data, length)
unsigned char *data;
int length;
{
	unsigned short crc = 0xffff;
	unsigned char *bufp = data;
	int len;
	static unsigned short bytetab[256] = {
		0x0000, 0x1189, 0x2312, 0x329b, 0x4624, 0x57ad, 0x6536, 0x74bf,
		0x8c48, 0x9dc1, 0xaf5a, 0xbed3, 0xca6c, 0xdbe5, 0xe97e, 0xf8f7,
		0x1081, 0x0108, 0x3393, 0x221a, 0x56a5, 0x472c, 0x75b7, 0x643e,
		0x9cc9, 0x8d40, 0xbfdb, 0xae52, 0xdaed, 0xcb64, 0xf9ff, 0xe876,
		0x2102, 0x308b, 0x0210, 0x1399, 0x6726, 0x76af, 0x4434, 0x55bd,
		0xad4a, 0xbcc3, 0x8e58, 0x9fd1, 0xeb6e, 0xfae7, 0xc87c, 0xd9f5,
		0x3183, 0x200a, 0x1291, 0x0318, 0x77a7, 0x662e, 0x54b5, 0x453c,
		0xbdcb, 0xac42, 0x9ed9, 0x8f50, 0xfbef, 0xea66, 0xd8fd, 0xc974,
		0x4204, 0x538d, 0x6116, 0x709f, 0x0420, 0x15a9, 0x2732, 0x36bb,
		0xce4c, 0xdfc5, 0xed5e, 0xfcd7, 0x8868, 0x99e1, 0xab7a, 0xbaf3,
		0x5285, 0x430c, 0x7197, 0x601e, 0x14a1, 0x0528, 0x37b3, 0x263a,
		0xdecd, 0xcf44, 0xfddf, 0xec56, 0x98e9, 0x8960, 0xbbfb, 0xaa72,
		0x6306, 0x728f, 0x4014, 0x519d, 0x2522, 0x34ab, 0x0630, 0x17b9,
		0xef4e, 0xfec7, 0xcc5c, 0xddd5, 0xa96a, 0xb8e3, 0x8a78, 0x9bf1,
		0x7387, 0x620e, 0x5095, 0x411c, 0x35a3, 0x242a, 0x16b1, 0x0738,
		0xffcf, 0xee46, 0xdcdd, 0xcd54, 0xb9eb, 0xa862, 0x9af9, 0x8b70,
		0x8408, 0x9581, 0xa71a, 0xb693, 0xc22c, 0xd3a5, 0xe13e, 0xf0b7,
		0x0840, 0x19c9, 0x2b52, 0x3adb, 0x4e64, 0x5fed, 0x6d76, 0x7cff,
		0x9489, 0x8500, 0xb79b, 0xa612, 0xd2ad, 0xc324, 0xf1bf, 0xe036,
		0x18c1, 0x0948, 0x3bd3, 0x2a5a, 0x5ee5, 0x4f6c, 0x7df7, 0x6c7e,
		0xa50a, 0xb483, 0x8618, 0x9791, 0xe32e, 0xf2a7, 0xc03c, 0xd1b5,
		0x2942, 0x38cb, 0x0a50, 0x1bd9, 0x6f66, 0x7eef, 0x4c74, 0x5dfd,
		0xb58b, 0xa402, 0x9699, 0x8710, 0xf3af, 0xe226, 0xd0bd, 0xc134,
		0x39c3, 0x284a, 0x1ad1, 0x0b58, 0x7fe7, 0x6e6e, 0x5cf5, 0x4d7c,
		0xc60c, 0xd785, 0xe51e, 0xf497, 0x8028, 0x91a1, 0xa33a, 0xb2b3,
		0x4a44, 0x5bcd, 0x6956, 0x78df, 0x0c60, 0x1de9, 0x2f72, 0x3efb,
		0xd68d, 0xc704, 0xf59f, 0xe416, 0x90a9, 0x8120, 0xb3bb, 0xa232,
		0x5ac5, 0x4b4c, 0x79d7, 0x685e, 0x1ce1, 0x0d68, 0x3ff3, 0x2e7a,
		0xe70e, 0xf687, 0xc41c, 0xd595, 0xa12a, 0xb0a3, 0x8238, 0x93b1,
		0x6b46, 0x7acf, 0x4854, 0x59dd, 0x2d62, 0x3ceb, 0x0e70, 0x1ff9,
		0xf78f, 0xe606, 0xd49d, 0xc514, 0xb1ab, 0xa022, 0x92b9, 0x8330,
		0x7bc7, 0x6a4e, 0x58d5, 0x495c, 0x3de3, 0x2c6a, 0x1ef1, 0x0f78,
	};

	for (len = length; len > 0; len--)
		crc = (crc >> 8) ^ bytetab[(crc & 0xff) ^ *bufp++];
	return crc;
}
```

Here is an intermediate version, doing the computation a nibble (4 bits) at a time, possibly useful on machines with small memories (or small fast caches):


```
/*
 - ssp2nibcrc - compute SSP2 CRC 4 bits at a time, quick and small table
 * Same parameters, algorithm, return value as the old nibcrc().
 */
unsigned short
ssp2nibcrc(data, length)
unsigned char *data;
int length;
{
	unsigned short crc = 0xffff;
	unsigned char *bufp = data;
	int len;
	static unsigned short lotab[16] = {
		0x0000, 0x1189, 0x2312, 0x329b, 0x4624, 0x57ad, 0x6536, 0x74bf,
		0x8c48, 0x9dc1, 0xaf5a, 0xbed3, 0xca6c, 0xdbe5, 0xe97e, 0xf8f7,
	};
	static unsigned short hitab[16] = {
		0x0000, 0x1081, 0x2102, 0x3183, 0x4204, 0x5285, 0x6306, 0x7387,
		0x8408, 0x9489, 0xa50a, 0xb58b, 0xc60c, 0xd68d, 0xe70e, 0xf78f,
	};

	for (len = length; len > 0; len--) {
		unsigned char ch = *bufp++ ^ crc;
		crc = (crc >> 8) ^ lotab[ch&0xf] ^ hitab[(ch&0xf0) >> 4];
	}
	return crc;
}
```

Finally, in case it's useful, here is the code that generates the tables for the byte and nibble versions:

```
/*
 - ssp2crctab - compute table for byte and nibble SSP2 CRCs
 * Uses ssp2bitcrc() to compute the CRC for each one-byte message, and
 * inverts ssp2bytecrc()'s algorithm to get the corresponding table entry.
 *
 * More subtly, the first 16 entries of the ssp2bytecrc() table are
 * ssp2nibcrc()'s low-order-nibble table, and every 16th entry is its
 * high-order-nibble table.
 */
void
ssp2crctab(tab)
unsigned short tab[256];	/* gets filled in */
{
	unsigned ch;
	unsigned char buf[1];
	unsigned short crc;

	for (ch = 0; ch <= 255; ch++) {
		buf[0] = ch;
		crc = ssp2bitcrc(buf, 1);
		tab[ch ^ 0xff] = crc ^ 0xff;
	}
}
```

