Jaeger System for Dev Ops Flight System Simulation

System Documentation

Revision 1

April 30, 2021

Introduction to the Jaeger System for Dev Ops Flight System Simulation (JS-DOFSS)

The Jaeger System Modernization Program is moving to microservices. The Jaeger System for Dev Ops Flight System Simulation (JS-DOFSS) is our first step towards modernizing the Jaeger System. The JS-DOFSS comprises a multi-device, RISC-V simulator capable of running multiple RISC-V microservices simultaneously (hereforth referred to as, "Devices"). The JS-DOFSS is a revolutionary design in Flight System design, providing unprecedented security at the system-bus level.

While many devices are planned, only a few are currently implemented. The devices currently available for demonstration are:

- The Access Control Device (ACD)
- The Maintenance Control Device (MCD)
- The RF Communications Device (RCD)
- The File Storage Device (FSD)

Additionally, the JS-DOFSS contains three separate busses. They are:

- The Privileged Bus.
- The Maintenance Bus.
- The RF Comms Bus.

A variety of external peripherals are available to devices within the JS-DOFSS. They are:

- The Bus Device
- The Debug Device
- The File Storage Device
- The Serial TCP Device
- The Test Device

Peripherals are memory-mapped into the the various devices. Because externally-driven interrupts are not yet implemented in JS-DOFSS, all devices run on regularly scheduled timer interrupts. Devices poll their periperhals for updated information, and interact with them according to their unique memory-mapped interfaces.

Finally, a variety of protocols are used between the JS-DOFSS systems. The, "Bus Protocol," is the standard protocol for inter-device communication. Each device normally presents its service via a unique protocol, which operates within the Bus Protocol. Those unique service protocols are:

- The Access Control Protocol.
- The RF Communications Protocol.
- The File Storage Protocol.

Devices

A diagram of the various devices, and their bus connections, is provided below:

```
                                   PriVileged Bus
++----------------+------+------------------------+
 |                |      |
 |                |      |         Maintenance Bus
+--+-------+---------------+----------------------+
 | |       |      |      | |
 | |       |      |      | |       RF Comms Bus
+-------------------+--------+--------------------+
 | |       |      | |    | | |
 | |       |      | |    | | |
 | |       |      | |    | | |
++-+--+ +--+--+ +-+-+-+ ++-+-++
| ACD | | MCD | | RCD | | FSD |
+-----+ +-----+ +-----+ +-----+
```

Access Control Device

System Configuration

- **Protocol Address**: 1
- **Connected Busses**: Privileged Bus, Maintenance Bus
- **Available Permissions**: None (N/A)

System Overview

The access control devices prevents unauthorized interactions between devices in the JS-DOFFS. It implements an authentication system, whereby a device can request a token from the ACD, and then use this token in requests to other devices. This token can be used to verify a requesting device has the correct permissions, without revealing the key of the requesting device.

All communications with the ACD take place on the Privileged Bus, except those communications between the ACD and the MCD, which take place on the Maintenance Bus.

The following permissions exist:

- `ACCESS_TYPE_FILESYSTEM` - Required to read/write files to the persistent data store.
- `ACCESS_TYPE_DEBUG` - Provides special debug operations to read/write memory, used for development and manufacturer-level maintenance.
- `ACCESS_TYPE_FLIGHT_CONTROLS` - Provides access to the flight controls and sensors. Can query and manipulate controls and sensors.
- `ACCESS_TYPE_COMMUNICATIONS` - Can send and receive messages via the communications system.
- `ACCESS_TYPE_MAINTENANCE` - Can perform device reprogramming.

Maintenance Control Device

System Configuration

- **Protocol Address**: 2
- **Connected Busses**: Maintenance Bus
- **Available Permissions**: All Permissions

System Overview

The maintenance device performs on-system checks to ensure the proper functioning of all systems on JS-DOFSS. These checks take place on a separate, isolated bus known as the maintenance bus. Further information about the functioning of the MCD, and its associated checks, can be found in JS-DOFSS MCD TM rev 3, and is not in scope for this document.

RF Communications Device (RCD)

System Configuration

- **Protocol Address**: 3
- **Connected Busses**: Privileged Bus, RF Comms Bus
- **Available Permissions**: `ACCESS_TYPE_COMMUNICATIONS`

System Overview

The RF Communications Device (RCD) provides programmatic off-platform access to some of the busses on the JS-DOFSS. For this demonstration, we have removed the RF Device and replaced it with a TCP/IP connection for testing purposes. While the RCD has the ability to both read and write messages to its connected busses, the primary purpose of the RCD is the enablement of off-platform bus monitoring for safety and maintenance purposes, as well as emergency systems control if required.

File Storage Device (FSD)

System Configuration

- **Protocol Address**: 4
- **Connected Busses**: Privileged Bus, Maintenance Bus, RF Comms Bus
- **Available Permissions**: `ACCESS_TYPE_COMMUNICATIONS`, `ACCESS_TYPE_FILESYSTEM`

System Overview

The File Storage Device provides authenticated access to sensitive files stored on the Jaeger System. All requests to the FSD require a token with the `ACCESS_TYPE_FILESYSTEM` permission. A basic flow chart of the FSD's functioning is provided below:

```
                                +------------------+
                                | Request Received |
                                +--------+---------+
                                         |
                                         v
                                 +-------+--------+  Invalid token  +----------------+
                                 | Validate Token +---------------->+ Return Failure |
                                 +-------+--------+                 +----------------+
                                         |
                                         | Valid Token
                                         v
                            +------------+-----------+
              +-------------+ Determine Request Type +------------+
              |             +-----------+------------+            |
              |                         |                         |
              | File Exists             | File Size               | File Contents
              |                         |                         |
              v                         v                         v
    +---------+----------+    +---------+----------+   +----------+-------------+
    | Does File Exist?   |    | Query Successful?  |   | Query Successful       |
    ++-----------+-------+    +-+------------------+   +-+-+--------------------+
     |           |            | |                        | |
     | Yes       | No         | | Yes                    | | Yes
     |           |            | | +-------------------+  | | +-----------------------------+
     v           v            | | | Return 1          |  | | | Return 1                    |
+----+-----+  +--+-------+    | +>+ Size field set to |  | | | Up to MAX bytes placed in   |
| Return 1 |  | Return 0 |    |   |   file size       |  | +>+   contents                  |
+----------+  +----------+    |   +-------------------+  |   | size set to number of bytes |
                              |                          |   |   placed in contents        |
                              | No                       |   +-----------------------------+
                              |   +----------+           |
                              +-->+ Return 0 |           | No
```

```
                                  +----------+           |   +----------+
                                                         +-->+ Return 0 |
                                                             +----------+
```

Protocols

Bus Protocol

Devices communicate with each other via the Bus Protocol. This protocol allows for up to 15 devices to communicate with either single-frame broadcast messages, single-frame data messages, or multi-frame streams.

Frame Format

Each frame consists of a two-byte header, followed by up to 6 data bytes.

Frame Header

The Frame Header consists of two bytes, the address byte and the flags byte. A breakout of these bytes, and the meaning of their bits is provided below.

```
Bit:   0123 4567 89AB CDEF
       ---- ---- ---- ----
Key:   FFFF TTTT BSDE IXXX

FFFF: 4 bits to represent the frame sender. This is the "From" field.
TTTT: 4 bits to represent the frame recipent. This is the "To" field.
B:    Broadcast Bit
S:    Stream Bit
D:    Data Bit
E:    Stream End Bit
I:    Stream Initiate Bit
XXX:  Size of frame data in bytes.
```

We can combine the values of several bits to represent different types of messages. An exhaustive list of message types, and the flag bits they must set, follows:

- Abort - Sent to a peer on protocol error, or when stream requests can not be satisfied.
- Broadcast - A frame sent to all peers on the bus.
- Stream - A frame which contains streaming data.
- Stream Ack - Acknowledgement of receipt of a Stream frame.
- Stream End - The last frame sent in a stream. Signals stream termination to the peer.
- Stream End Ack - An acknowledgement of a stream's end frame.
- Stream Initiate - Requests the initiation of a stream.
- Stream Initiate Ack - Acknowledges and accepts the inititiation of a stream.

```
                 Bit: 0123 4567
                 Key: BSDE IXXX
               ABORT: 1110 0000
           BROADCAST: 1000 0...
                DATA: 0010 0...
              STREAM: 0100 0...
          STREAM-ACK: 0110 0000
          STREAM-END: 0101 0...
      STREAM-END-ACK: 0111 0...
     STREAM_INITIATE: 0100 1100
STREAM-INITIATE-ACK: 0110 1...
```

Broadcasts

Broadcast messages are sent to address `0x0f` with the `BROADCAST` bit set, and are special frames that may be processed by all peers which support the processing of broadcast frames.

Broadcast frames are not acknowledged.

Data Frames

Data frames may be used when the amount of data to transmit is less than or equal to six bytes, and avoid the requirement to establish a stream.

Data frames are not acknowledged.

Streams

The protocol supports the streaming of up to `2^32-1` bytes, though peers while normally support streams of much smaller lengths.

We refer to the peer who initiates the stream as the `CLIENT`, and the peer who accepts and receives the stream as the `SERVER`. When the `STREAM` bit is set, the `DATA` bit is interpreted as an acknowledgement bit. The `STREAM-ACK` bits refer to both the `STREAM` and the `DATA` bits being set together.

Stream Initialization

Streams are initiated with a 6-byte frame. The address byte is filled in with `FROM` field set to the `CLIENT` address, and the `TO` field set to the `SERVER` address. The flags bit has the `STREAM` and `STREAM-INITIATE` bits set, with the data size field set to `4`. The intended length of the data to be streamed is then send in the following 4-bytes as a little-endian 32-bit value.

If the `SERVER` is willing to accept the stream, the client responds with the `STREAM-ACK`, and `STREAM-INITIATE` bits set.

If the `SERVER` is unwilling to accept the stream, the client responds with the `ABORT` bits set.

Streaming Data

Upon successful initialization of a stream, the `CLIENT` will send its data in consecutive 6-byte chunks until it has completed sending all of its data. If the `CLIENT` attempts to send more data than it has requested, the `SERVER` will respond with an `ABORT` frame, terminating the stream.

Each frame containing stream data will have the `STREAM` bit set, with the data size bits set to the number of data bytes available in the stream.

Each frame is acknowledged by the `SERVER` by setting the `STREAM-ACK` bits.

Ending the Stream

When the `CLIENT` is sending its last frame, it sets the `STREAM` and `STREAM-END` bits. Upon receiving a frame with the `STREAM` and `STREAM-END` bits set, the `SERVER` sends a response frame with the `STREAM-ACK` and `STREAM-END` bits set, successfully terminating the stream.

Access Control Protocol

The Access Control Protocol (ACP) works through streaming Bus Protocol messages, and provides interoperability between the ACD and other JS-DOFSS devices. The ACP exchanges 16-byte keys for 8-byte, randomly generated tokens, and allows other devices to verify permissions against those tokens.

A JS-DOFSS device sends the ACD its 16-byte key, along with its requested permissions, via an Access Token Creation Request. If the key is valid, and the key has access to the associated permissions, the ACD generates a random 8-byte token and returns it in an Access Token Creation Response. This token can be used in requests to other devices, proving the requesting device has the appropriate permissions without revealing the device's key.

This interaction is modeled in the below diagram:

```
Client               ACD                 Server
--------------------------------------------------
   |                   |                    |
Request Token--------->|                    |
   |             Validates Token            |
   |               Request                  |
   |                   |                    |
   |<-------------Issues Token              |
   |                   |                    |
Requests action        |                    |
with Token -------------------------------->|
   |                   |                    |
   |                   |<------------- Validates Token
   |                   |                    |
   |             Validates Token ---------->|
   |                   |                    |
   |                   |                Performs Action
   |<--------------------------------Sends Response
```

There are four message types implemented by the ACP. They are:

- Access Token Creation Request
- Access Token Creation Response
- Access Token Verification Request
- Access Token Verification Response

Each request and response begins with a 1-byte header, denoting the message type. The values for each message type are:

```
ACCESS_TOKEN_CREATE_REQUEST        = 0
ACCESS_TOKEN_CREATE_RESPONSE       = 1
ACCESS_TOKEN_VERIFICATION_REQUEST  = 2
ACCESS_TOKEN_VERIFICATION_RESPONSE = 3
```

Also important are the various values for access types. Access types denote the various permissions a key may hold, and a token is valid for. The access type is an 8-bit value with the following fields:

```
Bit Index:    0 1 2 3 4 5 6 7
Name:         F D U U C M U V

C: ACCESS_TYPE_COMMUNICATIONS
D: ACCESS_TYPE_DEBUG
F: ACCESS_TYPE_FILESYSTEM
M: ACCESS_TYPE_MAINTENANCE
U: Unused
V: ACCESS_TYPE_VALID
```

The access type is only valid if the `ACCESS_TYPE_VALID` bit is set. If the `ACCESS_TYPE_VALID` bit is set, the token has the permissions represented by the other set bits in the access type.

Access Token Creation Request

```
+---------------------+--------------------+-------------+
| 1-byte Request Type | 1-byte Acesss Type | 16-byte key |
+---------------------+--------------------+-------------+
```

The Access Token Creation Request consists of 1 byte, indicating the request type, 1-byte indiciating the permissions requested, at the 16-byte key from the requesting device.

The Request Type will be `ACCESS_TOKEN_CREATE_REQUEST`, and will always be set to `0`.

Access Token Creation Response

```
+----------------------+--------------------+--------------+
| 1-byte Response Type | 1-byte Acesss Type | 8-byte Token |
+----------------------+--------------------+--------------+
```

The Access Token Creation Response consists of 1 byte, indicating the response type, 1-byte indiciating the permissions requested, and the 8-byte generated token when the request is valid.

The Response Type will be `ACCESS_TOKEN_CREATE_RESPONSE`, and will always be set to `1`.

The access type will have the `ACCESS_TYPE_VALID` bit set if the request is valid, as well as the associated permissions. If the `ACCESS_TYPE_VALID` bit is set, the 8-byte token will be given in the token field. If the `ACCESS_TYPE_VALID` bit is not set, the 8-byte token field is undefined.

Access Token Verification Request

```
+---------------------+--------------------+--------------+
| 1-byte Request Type | 1-byte Acesss Type | 8-byte Token |
+---------------------+--------------------+--------------+
```

The Access Token Verification Request consists of 1 byte, indicating the request type, 1-byte indiciating the permissions which need to be verified, and the 8-byte token to verify.

The Request Type will be `ACCESS_TOKEN_VERIFICATION_REQUEST`, and will always be set to `2`.

Access Token Verification Response

```
+----------------------+--------------------+--------------+
| 1-byte Response Type | 1-byte Acesss Type | 8-byte Token |
+----------------------+--------------------+--------------+
```

The Access Token Verification Response consists of 1 byte, indicating the response type, 1-byte indiciating the verified permissions, and the 8-byte token which was verified.

The Response Type will be `ACCESS_TOKEN_VERIFICATION_RESPONSE`, and will always be set to `3`.

If the requested access type is valid for the token, the requested permissions will be returned in the access type with the `ACCESS_TYPE_VALID` bit set. The 8-byte token field will always return the token given in the request.

RF Communications Protocol

The RF Communications Protocol (RCP) allows devices external to the JF-DOFSS to communicate with the RCD via an RF comms link. For the purposes of this demonstration, the RF comms link has been replaced with a TCP/IP connection.

The RCP allows for the relaying of messages to and from busses connected to the RF Comms device, as well as injecting arbitrary bitstreams onto the currently relayed bus.

There are four RCP commands, given below:

```
DISABLE_COMMS_RELAY  = 0xF0
RELAY_PRIVILEGED_BUS = 0xF1
RELAY_RF_COMMS_BUS   = 0xF3
RELAY_MESSAGE_TO_BUS = 0xA0
```

These commands allow the RCD to be put in one of three states:

- Not relaying messages.
- Relaying messages from the Privileged Cus.
- Relaying messages from the RF Comms Bus.

If the RCD is relaying messages from a particular bus, the `RELAY_MESSAGE_TO_BUS` command allows the `RCD` to send messages to that bus.

The RCD will always be initially placed in the state where it is not relaying bus messages.

Disable Comms Relay

The disable comms relay command is a 1-byte command sent to the RCD. Sending the byte `0xF0` to the RCD disables all message relaying.

Relay Privileged Bus

The relay privileged bus command is a 1-byte command send to the RCD. Sending the byte `0xF1` to the RCD will cause the RCD to begin relaying all messages sent over the privileged bus, as well as allow users to inject messages into the privileged bus via the `RELAY_MESSAGE_TO_BUS` command.

Relay RF Comms

The relay RF comms bus command is a 1-byte command send to the RCD. Sending the byte `0xF3` to the RCD will cause the RCD to begin relaying all messages sent over the privileged bus, as well as allow users to inject messages into the privileged bus via the `RELAY_MESSAGE_TO_BUS` command.

Relay Messages to Bus

The relay messages bus command is formatted as follows:

```
+----------------+---------------+-------------------------+
| 1-Byte Command | 1-Byte Length | Variable Length Message |
+----------------+---------------+-------------------------+
```

The 1-byte command will always be `0xA0`. The length field designates the number of bytes in the message field that should be written to the currently relayed bus. When processing this messages, the RCD will copy the `length` bytes from the `message` field verbatim onto the currently relayed bus.

THIS COMMAND IS ONLY INTENDED TO BE USED IN EMERGENCY SITUATIONS

File Storage Protocol

The File Storage Protocol (FSP) works through stream Bus Protocol messages, and provides interoperability between the FSD and other JS-DOFSS devices. The FSP allows devices with the `ACCESS_TYPE_FILESYSTEM` permission to read sensitive files stored by the FSD.

A JS-DOFSS device can request the existence of files, the size of files, and the contents of files. A typical exchange with the FSD is modelled in the below diagram:

```
+------+                    +---+                      +---+
|Device|                    |FSD|                      |ACD|
+------+                    +-+-+                      +-+-+
   |                          |                          |
   +-------------------------+---------------------------->+
   |              Token Creation Request                   |
   |                         |                              |
   +<------------------------+-----------------------------+
   |              Token Creation Response                  |
   |                         |                              |
   +------------------------>+                              |
   |      FSD Request        |                              |
   |                         |                              |
   |                         +---------------------------->+
   |                         | Token Verification Request  |
   |                         |                              |
   |                         +<----------------------------+
   |                         | Token Verification Response |
   |                         |                              |
   +<------------------------+                              |
   |      FSD Response       |                              |
   +                         +                              +
```

There are six message types implemented by the FSP. They are:

- File Exists Request
- File Exists Response
- File Size Request
- File Size Response
- File Contents Request
- File Contents Response

Each request begins with a 1-byte message type, followed by an 8-byte token. Each response begins with a 1-byte type. The available types are:

```
FSD_PROT_FILE_EXISTS_REQUEST    = 1
FSD_PROT_FILE_SIZE_REQUEST      = 2
FSD_PROT_FILE_CONTENTS_REQUEST  = 3
FSD_PROT_FILE_EXISTS_RESPONSE   = 4
FSD_PROT_FILE_SIZE_RESPONSE     = 5
FSD_PROT_FILE_CONTENTS_RESPONSE = 6
```

This protocol has two constants, `FSD_FILENAME_SIZE` and `FSD_CONTENTS_SIZE`.

```
FSD_FILENAME_SIZE = 0x80
FSD_CONTENTS_SIZE = 0x40
```

File Exists Request

```
+---------------------+-------------------------+
| 1-byte Request Type | Variable-length filename |
+---------------------+-------------------------+
```

The file exists request begins with the byte `FSD_PROT_FILE_EXISTS_REQUEST`, `0x01`, followed by a null-terminated string up to `0x80` bytes in length. If the filename is not null-terminated, the FSD will null-terminate the filename for you.

File Exists Response

```
+----------------------+-----------------+
| 1-byte Response Type | 1-byte Response |
+----------------------+-----------------+
```

The file exists response begins with the byte `FSD_PROT_FILE_EXISTS_RESPONSE`, `0x02`, followed by a single byte which indicates whether the file exists. If the response byte is `0x01`, the file exists. If the response byte is `0x00`, the file does not exist.

File Size Request

```
+---------------------+--------------------------+
| 1-byte Request Type | Variable-length filename |
+---------------------+--------------------------+
```

The file exists request begins with the byte `FSD_PROT_FILE_SIZE_REQUEST`, `0x03`, followed by a null-terminated string up to `0x80` bytes in length. If the filename is not null-terminated, the FSD will null-terminate the filename for you.

File Size Response

```
+----------------------+----------------+--------------+
| 1-byte Response Type | 1-byte Success | 4-byte Size |
+----------------------+----------------+--------------+
```

The file size response begins with the byte `FSD_PROT_FILE_SIZE_RESPONSE`, `0x04`, followed by a 1-byte success value, followed by the file size in a 32-bit little-endian field. If the success value is `1`, the `size` field contains the size of the requested file. If the success value is `0`, the `size` field is undefined.

File Contents Request

```
+---------------------+---------------+-------------+--------------------------+
| 1-byte Request Type | 4-byte Offset | 4-byte Size | Variable-length filename |
+---------------------+---------------+-------------+--------------------------+
```

The file contents request begins with a 1-byte request type, followed by a 4-byte little-endian offset from the beginning file, followed by a 4-byte little-endian size field for the number of bytes requested, and trailed by a null-terminated filename with a max-length of `0x80` bytes. If the filename is not null-terminated, the FCD will null-terminate the filename for you.

This request is meant to be sent multiple times, requesting subsequent chunks of a file, until the entire file has been fetched from the FSD.

The maximum value for the `size` field is `0x40` bytes. If `size` is greater than `0x40` bytes, the File Storage Device will truncate the response to `0x40` bytes.

File Contents Response

```
+----------------------+----------------+---------------+-------------+
| 1-byte Response Type | 1-byte Success | 4-byte Offset | 4-byte Size |
+----------------------+---+------------+---------------+-------------+
| Variable-length contents |
+--------------------------+
```

The file contents response begins with a 1-byte response type, followed by a 4-byte little endian offset from the beginning of the file, followed by a 4-byte little-endian size field for the number of bytes returned in contents, followed by a variable-length array containing file contents. The number of bytes returned in the `contents` field is determined by the `size` field, and will not exceed `FSD_CONTENTS_SIZE`, which is set to `0x40`.

If the response type is `1`, the other field in this response type are valid. If the response type is `0`, `offset` should be `0`, `size` should be `0`, and `contents` should be omitted.

Peripherals

The JS-DOFSS is responsible for mapping perhipherals into device memory. The JS-DOFSS supports the mapping of up to 8 peripherals per device. Devices are mapped into device memory beginning at address `0x40000000`, and increment by `0x00010000` bytes per device.

```
DEVICE 0 ADDRESS: 0x40000000
DEVICE 1 ADDRESS: 0x40010000
DEVICE 2 ADDRESS: 0x40020000
DEVICE 3 ADDRESS: 0x40030000
DEVICE 4 ADDRESS: 0x40040000
DEVICE 5 ADDRESS: 0x40050000
DEVICE 6 ADDRESS: 0x40060000
DEVICE 7 ADDRESS: 0x40070000
```

Which peripheral type is mapped to which address is dependent on the device.

There are 6 device types. They are:

- Bus Device
- Debug Device
- Dummy Device
- File Storage Device
- Serial TCP Device
- Test Device

Each peripheral serves a unique purpose, and provides a unique memory mapping to a device.

While noted previously, we note again that JS-DOFSS does not yes currently support externally-driven interrupts. All devices run off internal timer-driven interrupts. Devices will poll peripherals for information on a regular interval.

Bus Peripheral

The bus peripheral allows devices to read and write bytes to common busses. Bytes written to busses can be read from, and written to, by all devices connected to the bus. The bus provides the sole means for inter-device communication.

Overview

```
+--------+------+--------------+
| Offset | Size | Field        |
+--------+------+--------------+
| 0x0000 | 0x01 | STATUS       |
+--------+------+--------------+
| 0x0001 | 0x01 | READ_SIZE    |
+--------+------+--------------+
| 0x0002 | 0x01 | WRITE_SIZE   |
+--------+------+--------------+
| 0x0100 | 0x10 | READ BUFFER  |
+--------+------+--------------+
| 0x0200 | 0x10 | WRITE BUFFER |
+--------+------+--------------+
```

Functioning

The bus device is controlled through the STATUS bit. The STATUS bit has the following fields:

```
Bit Index:    0 1 2 3 4 5 6 7
Name:         R C W D U U U U

R: READ DATA READY BIT
C: READ DATA COMPLETE BIT
W: WRITE DATA READY BIT
D: WRITE DATA COMPLETE BIT
U: UNUSED
```

- `READ_DATA_READY_BIT`: When this bit is set, the bus peripheral has data available in the read buffer the device should read.
- `READ_DATA_COMPLETE_BIT`: When this bit is set, the device has finished reading data from the read buffer. The device must also set the `READ_DATA_READY_BIT` to `0`.
- `WRITE_DATA_READY_BIT`: When this bit is set, the device has placed data into the `WRITE_BUFFER`, and the bus device should send that data to the bus.
- `WRITE_DATA_COMPLETE_BIT`: When this bit is set, the bus peripheral has written the data from the write buffer to the bus. Also, the bus peripheral must set the `WRITE_DATA_READY_BIT` to `0`.

At all times the `READ_DATA_READY_BIT` and `WRITE_DATA_READY_BIT` are set, the `WRITE_SIZE` and `READ_SIZE` fields be valid for the number of bytes to be read and written from teh `READ_BUFFER` and `WRITE_BUFFER` fields.

Debug Peripheral

The debug peripheral provides a mechanism for devices to provide debugging information by means of printing strings and integers. The output of debug peripherals should display to `STDOUT` for users of the `JS-DOFSS`.

Overview

Offset	Size	Field
0x0000	0x01	WRITE
0x0100	0x100	BUF

Acceptable values for `WRITE` are:

```
SEND_STRING = 1
SEND_UINT8  = 2
SEND_UINT32 = 3
```

Functioning

A device can write one of three values to the `WRITE` field. The interpretation of the `BUF` field is dependent upon the value of the `WRITE` field.

When the `WRITE` field is set to `STRING`, a null-terminated string will be read from the `BUF` field and written to `STDOUT`.

When the `WRITE` field is set to `UINT8`, a single byte will be read from the `BUF` field and written to `STDOUT`.

When the `WRITE` field is set to `UINT32`, a 32-bit little-endian integer will be read from the `BUF` field and written to `STDOUT`.

Dummy Peripheral

Certain device libraries require peripherals to be present at certain addresses. A “Dummy Device” is a peripheral for which all reads and writes are invalid, but allows the JS-DOFSS to advance advance peripheral addresses. We can insert dummy devices to force other peripherals to exist at specific addresses.

File Storage Peripheral

The file storage peripheral provides a device access to the underlying filesystem.

Overview

Offset	Size	Field
0x0000	0x01	ACTION
0x0004	0x01	SIZE
0x0008	0x04	OFFSET
0x0080	0x80	FILENAME
0x0100	0x40	CONTENTS

Functioning

The File Storage Peripheral takes three commands. They are:

```
FSD_ACTION_FILE_EXISTS = 1
FSD_ACTION_FILE_SIZE   = 2
FSD_ACTION_READ_FILE   = 3
```

File Exists

The file exists action checks for the existence of a given file.

To use this action, first place the filename of the file you intend to check into the `FILENAME` field as a null-terminated string. Then write the value `0x01` for `FSD_ACTION_FILE_EXISTS` into the `ACTION` register.

The peripheral will cause the device to block until the action is complete. Once complete, the device can read from the `ACTION` register. If the `ACTION` contains 1, the file exists. If the `ACTION` register contains `0`, the file does not exist.

File Size

The file size action returns the size of a file.

To use this action, first place the filename of the file you intend to check into the `FILENAME` field as a null-terminated string. Then write the value `0x02` for `FSD_ACTION_FILE_SIZE` into the `ACTION` register.

The peripheral will cause the device to block until the action is complete. Once complete, the device can read from the `ACTION` register. If the `ACTION` register contains `1`, the file size can be read from the `SIZE` field. If the `ACTION` register contains `0`, the peripheral failed to get the size of the requested file.

File Size

The file size action returns the size of a file.

To use this action, first place the filename of the file you intend to check into the `FILENAME` field as a null-terminated string. Then write the value `0x02` for `FSD_ACTION_FILE_SIZE` into the `ACTION` register.

The peripheral will cause the device to block until the action is complete. Once complete, the device can read from the `ACTION` register. If the `ACTION` register contains `1`, the file size can be read from the `SIZE` field. If the `ACTION` register contains `0`, the peripheral failed to get the size of the requested file.

File Contents

The file contents actions fetches up to 64 bytes from a file at a given offset.

To use this action, first place the filename of the file for which you want contents into the `FILENAME` field as a null-terminated string. Place the number of bytes you wish to read into the `SIZE` field. Place the offset from the beginning of the file where you want the read to take place into the `OFFSET` field. Then write the value `0x03` for `FSD_ACTION_READ_FILE` into the `ACTION` register.

Both the `SIZE` and `OFFSET` fields take 32-bit little-endian values. If the value written to the `SIZE` field is greater than `0x40`, only upto `0x40` bytes will be read.

The peripheral will cause the device to block until the action is complete. Once complete, the device can read from the `ACTION` register.

If the `ACTION` register contains `1`, the requested file contents can be found in the `CONTENTS` field. The `SIZE` field will denote how many bytes were read from the `FILE`. The value of the `CONTENTS` field beyond the bounds denoted by the `SIZE` field are undetermined.

If the `ACTION` register contains `0`, the requested read failed and the value of all other fields is undefined.

Test Peripheral

The test peripheral allows devices to signal when the system appears to be malfunctioning. The test peripheral is used primarily by the maintenance device during its startup checks.

Overview

```
+--------+-------+--------------+
| Offset | Size  | Field        |
+--------+-------+--------------+
| 0x0000 | 0x01  | TEST FAIL    |
+--------+-------+--------------+
| 0x0001 | 0x01  | TEST SUCCESS |
+--------+-------+--------------+
```

Functioning

The test peripheral works very simply.

Any write to the `TEST FAIL` register will send an error message to `STDOUT` and cause the JS-DOFSS to exit.

Any write to the `TEST SUCCESS` register will send a message to `STDOUT`. JS-DOFSS will continue to function normally.