MessagePorts in Electron

`MessagePort` s are a web feature that allow passing messages between different contexts. It's like `window.postMessage`, but on different channels. The goal of this document is to describe how Electron extends the Channel Messaging model, and to give some examples of how you might use MessagePorts in your app.

Here is a very brief example of what a MessagePort is and how it works:

```
// renderer.js ////////////////////////////////////////////////////////////////
// MessagePorts are created in pairs. A connected pair of message ports is
// called a channel.
const channel = new MessageChannel()

// The only difference between port1 and port2 is in how you use them. Messages
// sent to port1 will be received by port2 and vice-versa.
const port1 = channel.port1
const port2 = channel.port2

// It's OK to send a message on the channel before the other end has registered
// a listener. Messages will be queued until a listener is registered.
port2.postMessage({ answer: 42 })

// Here we send the other end of the channel, port1, to the main process. It's
// also possible to send MessagePorts to other frames, or to Web Workers, etc.
ipcRenderer.postMessage('port', null, [port1])
```

```
// main.js ////////////////////////////////////////////////////////////////////
// In the main process, we receive the port.
ipcMain.on('port', (event) => {
  // When we receive a MessagePort in the main process, it becomes a
  // MessagePortMain.
  const port = event.ports[0]

  // MessagePortMain uses the Node.js-style events API, rather than the
  // web-style events API. So .on('message', ...) instead of .onmessage = ...
  port.on('message', (event) => {
    // data is { answer: 42 }
    const data = event.data
  })

  // MessagePortMain queues messages until the .start() method has been called.
  port.start()
})
```

The Channel Messaging API documentation is a great way to learn more about how MessagePorts work.

MessagePorts in the main process

In the renderer, the `MessagePort` class behaves exactly as it does on the web. The main process is not a web page, though—it has no Blink integration—and so it does not have the `MessagePort` or `MessageChannel`

classes. In order to handle and interact with MessagePorts in the main process, Electron adds two new classes: `MessagePortMain` and `MessageChannelMain`. These behave similarly to the analogous classes in the renderer.

`MessagePort` objects can be created in either the renderer or the main process, and passed back and forth using the `ipcRenderer.postMessage` and `WebContents.postMessage` methods. Note that the usual IPC methods like `send` and `invoke` cannot be used to transfer `MessagePort`s, only the `postMessage` methods can transfer `MessagePort`s.

By passing `MessagePort`s via the main process, you can connect two pages that might not otherwise be able to communicate (e.g. due to same-origin restrictions).

Extension: `close` event

Electron adds one feature to `MessagePort` that isn't present on the web, in order to make MessagePorts more useful. That is the `close` event, which is emitted when the other end of the channel is closed. Ports can also be implicitly closed by being garbage-collected.

In the renderer, you can listen for the `close` event either by assigning to `port.onclose` or by calling `port.addEventListener('close', ...)`. In the main process, you can listen for the `close` event by calling `port.on('close', ...)`.

Example use cases

Worker process

In this example, your app has a worker process implemented as a hidden window. You want the app page to be able to communicate directly with the worker process, without the performance overhead of relaying via the main process.

```
// main.js ////////////////////////////////////////////////////////////////////
const { BrowserWindow, app, ipcMain, MessageChannelMain } = require('electron')

app.whenReady().then(async () => {
  // The worker process is a hidden BrowserWindow, so that it will have access
  // to a full Blink context (including e.g. <canvas>, audio, fetch(), etc.)
  const worker = new BrowserWindow({
    show: false,
    webPreferences: { nodeIntegration: true }
  })
  await worker.loadFile('worker.html')

  // The main window will send work to the worker process and receive results
  // over a MessagePort.
  const mainWindow = new BrowserWindow({
    webPreferences: { nodeIntegration: true }
  })
  mainWindow.loadFile('app.html')

  // We can't use ipcMain.handle() here, because the reply needs to transfer a
  // MessagePort.
```

```
  ipcMain.on('request-worker-channel', (event) => {
    // For security reasons, let's make sure only the frames we expect can
    // access the worker.
    if (event.senderFrame === mainWindow.webContents.mainFrame) {
      // Create a new channel ...
      const { port1, port2 } = new MessageChannelMain()
      // ... send one end to the worker ...
      worker.webContents.postMessage('new-client', null, [port1])
      // ... and the other end to the main window.
      event.senderFrame.postMessage('provide-worker-channel', null, [port2])
      // Now the main window and the worker can communicate with each other
      // without going through the main process!
    }
  })
})
```

```
<!-- worker.html ------------------------------------------------------------->
<script>
const { ipcRenderer } = require('electron')

const doWork = (input) => {
  // Something cpu-intensive.
  return input * 2
}

// We might get multiple clients, for instance if there are multiple windows,
// or if the main window reloads.
ipcRenderer.on('new-client', (event) => {
  const [ port ] = event.ports
  port.onmessage = (event) => {
    // The event data can be any serializable object (and the event could even
    // carry other MessagePorts with it!)
    const result = doWork(event.data)
    port.postMessage(result)
  }
})
</script>
```

```
<!-- app.html ---------------------------------------------------------------->
<script>
const { ipcRenderer } = require('electron')

// We request that the main process sends us a channel we can use to
// communicate with the worker.
ipcRenderer.send('request-worker-channel')

ipcRenderer.once('provide-worker-channel', (event) => {
  // Once we receive the reply, we can take the port...
  const [ port ] = event.ports
  // ... register a handler to receive results ...
```

```
  port.onmessage = (event) => {
    console.log('received result:', event.data)
  }
  // ... and start sending it work!
  port.postMessage(21)
})
</script>
```

Reply streams

Electron's built-in IPC methods only support two modes: fire-and-forget (e.g. `send`), or request-response (e.g. `invoke`). Using MessageChannels, you can implement a "response stream", where a single request responds with a stream of data.

```
// renderer.js ////////////////////////////////////////////////////////////////

const makeStreamingRequest = (element, callback) => {
  // MessageChannels are lightweight--it's cheap to create a new one for each
  // request.
  const { port1, port2 } = new MessageChannel()

  // We send one end of the port to the main process ...
  ipcRenderer.postMessage(
    'give-me-a-stream',
    { element, count: 10 },
    [port2]
  )

  // ... and we hang on to the other end. The main process will send messages
  // to its end of the port, and close it when it's finished.
  port1.onmessage = (event) => {
    callback(event.data)
  }
  port1.onclose = () => {
    console.log('stream ended')
  }
}

makeStreamingRequest(42, (data) => {
  console.log('got response data:', event.data)
})
// We will see "got response data: 42" 10 times.
```

```
// main.js ////////////////////////////////////////////////////////////////////

ipcMain.on('give-me-a-stream', (event, msg) => {
  // The renderer has sent us a MessagePort that it wants us to send our
  // response over.
  const [replyPort] = event.ports
```

```
  // Here we send the messages synchronously, but we could just as easily store
  // the port somewhere and send messages asynchronously.
  for (let i = 0; i < msg.count; i++) {
    replyPort.postMessage(msg.element)
  }

  // We close the port when we're done to indicate to the other end that we
  // won't be sending any more messages. This isn't strictly necessary--if we
  // didn't explicitly close the port, it would eventually be garbage
  // collected, which would also trigger the 'close' event in the renderer.
  replyPort.close()
})
```

Communicating directly between the main process and the main world of a context-isolated page

When context isolation is enabled, IPC messages from the main process to the renderer are delivered to the isolated world, rather than to the main world. Sometimes you want to deliver messages to the main world directly, without having to step through the isolated world.

```
// main.js ////////////////////////////////////////////////////////////////////
const { BrowserWindow, app, MessageChannelMain } = require('electron')
const path = require('path')

app.whenReady().then(async () => {
  // Create a BrowserWindow with contextIsolation enabled.
  const bw = new BrowserWindow({
    webPreferences: {
      contextIsolation: true,
      preload: path.join(__dirname, 'preload.js')
    }
  })
  bw.loadURL('index.html')

  // We'll be sending one end of this channel to the main world of the
  // context-isolated page.
  const { port1, port2 } = new MessageChannelMain()

  // It's OK to send a message on the channel before the other end has
  // registered a listener. Messages will be queued until a listener is
  // registered.
  port2.postMessage({ test: 21 })

  // We can also receive messages from the main world of the renderer.
  port2.on('message', (event) => {
    console.log('from renderer main world:', event.data)
  })
  port2.start()

  // The preload script will receive this IPC message and transfer the port
  // over to the main world.
```

```
  bw.webContents.postMessage('main-world-port', null, [port1])
})
```

```
// preload.js ///////////////////////////////////////////////////////////////
const { ipcRenderer } = require('electron')

// We need to wait until the main world is ready to receive the message before
// sending the port. We create this promise in the preload so it's guaranteed
// to register the onload listener before the load event is fired.
const windowLoaded = new Promise(resolve => {
  window.onload = resolve
})

ipcRenderer.on('main-world-port', async (event) => {
  await windowLoaded
  // We use regular window.postMessage to transfer the port from the isolated
  // world to the main world.
  window.postMessage('main-world-port', '*', event.ports)
})
```

```
<!-- index.html ------------------------------------------------------------->
<script>
window.onmessage = (event) => {
  // event.source === window means the message is coming from the preload
  // script, as opposed to from an <iframe> or other source.
  if (event.source === window && event.data === 'main-world-port') {
    const [ port ] = event.ports
    // Once we have the port, we can communicate directly with the main
    // process.
    port.onmessage = (event) => {
      console.log('from main process:', event.data)
      port.postMessage(event.data * 2)
    }
  }
}
</script>
```