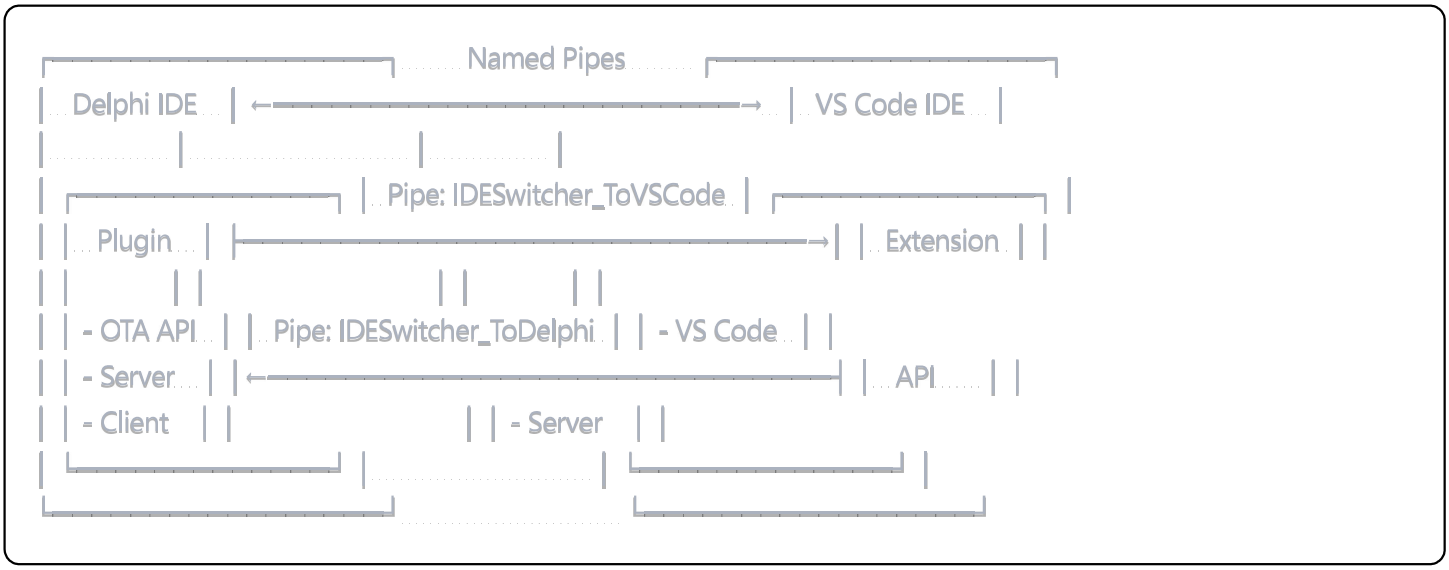
IDE Switcher - Implementation Guidelines

Architecture Overview

System Design

The IDE Switcher uses a **bidirectional client-server architecture** where both IDEs act as both client and server:



Communication Protocol

Message Format (JSON)

```json
{
  "action": "switch",
  "filePath": "C:\\Projects\\MyApp\\Unit1.pas",
  "line": 42,
  "column": 15,
  "source": "delphi|vscode",
  "timestamp": "2024-01-15T10:30:00Z",
  "version": "1.0"
}
```

Named Pipes

- **Pipe Names**:
  - `\\.\pipe\IDESwitcher_ToVSCode` (Delphi → VS Code)
  - `\\.\pipe\IDESwitcher_ToDelphi` (VS Code → Delphi)
- **Type**: Message-mode pipes (not byte-stream)
- **Security**: Local session only (current user)

- **Buffer Size**: 4096 bytes (sufficient for JSON messages)

Delphi Plugin Implementation

Core Components

1. Open Tools API Integration

```pascal
IOTAWizard          // Main plugin interface
IOTAKeyboardBinding // Keyboard shortcut handler
IOTAModuleServices  // File and module access
IOTAEditView        // Cursor position control
IOTAActionServices  // File operations
```

2. Key Classes

TIDESwitcherWizard

- Purpose: Main plugin entry point
- Responsibilities:
  - Register with IDE
  - Create keyboard binding
  - Start pipe server thread
  - Cleanup on exit

TIDESwitcherNotifier

- Purpose: Handle keyboard events
- Responsibilities:
  - Respond to Ctrl+Shift+D
  - Get current file info
  - Save file
  - Send to VS Code

TPipeServerThread

- Purpose: Listen for VS Code requests
- Responsibilities:
  - Maintain pipe server
  - Parse incoming JSON
  - Open files in Delphi

- Position cursor

Implementation Details

Getting Current File Info

```pascal
ModuleServices := BorlandIDEServices as IOTAModuleServices;
Module := ModuleServices.CurrentModule;
Editor := Module.CurrentEditor;
SourceEditor := Editor as IOTASourceEditor;
EditView := SourceEditor.GetEditView(0);
EditPos := EditView.CursorPos; // Line and column
```

Opening Files

```pascal
ActionServices := BorlandIDEServices as IOTAActionServices;
ActionServices.OpenFile(FilePath);
// Then position cursor using EditView.CursorPos
```

Thread Safety

- Pipe server runs in separate thread
- Uses `TThread.Synchronize` for IDE operations
- All OTA calls must be in main thread

Error Handling

1. **Pipe Connection Failures**
   - Retry mechanism (3 attempts)
   - Fallback to command-line launch
   - User notification
2. **File Operation Failures**
   - Validate file existence
   - Handle read-only files
   - Check module state before save
3. **Position Errors**
   - Bounds checking for line/column
   - Default to position 1,1 if invalid

- Handle different file encodings

VS Code Extension Implementation

Core Components

1. Extension Architecture

```typescript
activate()  // Extension entry point
deactivate()  // Cleanup
registerCommand() // Keyboard binding
vscode.window.activeTextEditor // Current file access
vscode.workspace.openTextDocument() // File operations
```

2. Key Functions

switchToDelphi()

- Save current file
- Get file path and position
- Send to Delphi via pipe
- Handle connection errors

startPipeServer()

- Create named pipe server
- Listen for Delphi requests
- Auto-restart on failure

handleDelphiRequest()

- Parse incoming JSON
- Open requested file
- Position cursor
- Focus VS Code window

Implementation Details

Getting Current Position

```typescript
```

```
const editor = vscode.window.activeTextEditor;
const position = editor.selection.active;
const line = position.line + 1; // Convert to 1-based
const column = position.character + 1;
```

Opening Files

```typescript
const document = await vscode.workspace.openTextDocument(filePath);
const editor = await vscode.window.showTextDocument(document);
editor.selection = new vscode.Selection(position, position);
editor.revealRange(range, vscode.TextEditorRevealType.InCenter);
```

Named Pipe Client/Server

```typescript
// Server
const server = net.createServer((client) => {
  client.on('data', (chunk) => { /* handle */ });
});
server.listen('\\\\.\\pipe\\PipeName');

// Client
const client = net.createConnection('\\\\.\\pipe\\PipeName');
client.write(JSON.stringify(data));
```

Error Handling

1. **Pipe Failures**
   - Automatic retry with backoff
   - PowerShell fallback for window activation
   - User notifications
2. **File Operations**
   - Handle missing files gracefully
   - Support various encodings
   - Manage workspace-relative paths

Inter-Process Communication

Named Pipes Details

Why Named Pipes?

- **Fast**: Shared memory implementation
- **Reliable**: Message boundaries preserved
- **Secure**: Windows access control
- **Simple**: No firewall issues

Alternative Considered

- **TCP/IP**: Firewall issues, port conflicts
- **File watching**: Too slow, polling required
- **Windows Messages**: Limited data size
- **COM/OLE**: Complex, version dependencies

Message Flow

Delphi → VS Code

1. User presses Ctrl+Shift+D in Delphi
2. Plugin saves file via OTA
3. Plugin gets file path and cursor position
4. Plugin connects to VS Code's pipe
5. Plugin sends JSON message
6. VS Code receives and opens file
7. VS Code positions cursor and focuses

VS Code → Delphi

1. User presses Ctrl+Shift+D in VS Code
2. Extension saves file via API
3. Extension gets file path and position
4. Extension connects to Delphi's pipe
5. Extension sends JSON message
6. Delphi receives and opens file
7. Delphi positions cursor and focuses

Synchronization

- **No locking required**: Each pipe is unidirectional
- **No state sharing**: Each IDE maintains own state
- **Idempotent operations**: Repeated switches are safe
- **Timeout handling**: 5-second timeout on connections

Platform Considerations

Windows-Specific Features

Named Pipes

```pascal
CreateNamedPipe(
  PChar(PipeName),
  PIPE_ACCESS_DUPLEX,
  PIPE_TYPE_MESSAGE or PIPE_READMODE_MESSAGE,
  PIPE_UNLIMITED_INSTANCES,
  BufferSize, BufferSize, 0, nil
);
```

Window Management

```pascal
SetForegroundWindow(Handle);
ShowWindow(Handle, SW_RESTORE);
```

Compatibility

Delphi Versions

- **Minimum**: XE2 (for modern OTA)
- **Tested**: XE7, 10.x, 11.x, 12.x
- **Notes**: Adjust unit names for older versions

VS Code Versions

- **Minimum**: 1.74.0
- **Works with**: WindSurf, Cursor, other forks
- **API**: Uses stable extension API only

Performance Optimization

Speed Targets

- **Switch time**: < 200ms typical
- **File save**: < 50ms for normal files
- **Pipe communication**: < 10ms local
- **Window activation**: < 100ms

Optimization Techniques

1. **Lazy Initialization**
   - Don't create pipes until needed
   - Cache window handles
   - Reuse pipe connections
2. **Async Operations**
   - Non-blocking pipe I/O
   - Async file operations in VS Code
   - Thread pool in Delphi
3. **Resource Management**
   - Close pipes immediately after use
   - Dispose of JSON objects
   - Clear message buffers

Security Considerations

Pipe Security

- **Local only**: Pipes use "\.\pipe" prefix
- **User session**: Not accessible across sessions
- **No network access**: Cannot be accessed remotely
- **ACL protection**: Windows default security

Data Protection

- **No sensitive data**: Only file paths and positions
- **No code transmission**: File contents stay local
- **No external connections**: Everything is local
- **No logging by default**: Debug logs only in dev mode

Testing Guidelines

Unit Tests

Delphi Plugin

```pascal
procedure TestGetCurrentFile;
procedure TestSaveFile;
procedure TestJSONSerialization;
procedure TestPipeConnection;
procedure TestCursorPosition;
```

VS Code Extension

```typescript
suite('IDE Switcher Tests', () => {
  test('Get current file', async () => {});
  test('Save file', async () => {});
  test('JSON serialization', () => {});
  test('Pipe connection', async () => {});
  test('Cursor position', async () => {});
});
```

Integration Tests

1. **Basic Switch Test**
   - Open file in Delphi
   - Press Ctrl+Shift+D
   - Verify VS Code opens same file
   - Verify cursor position
2. **Bidirectional Test**
   - Switch from Delphi to VS Code
   - Make changes
   - Switch back to Delphi
   - Verify changes preserved
3. **Error Handling Test**
   - Test with missing files
   - Test with read-only files
   - Test with no connection

- Test with invalid positions

Performance Tests

```typescript
// Measure switch time
const startTime = Date.now();
await switchToDelphi();
const switchTime = Date.now() - startTime;
assert(switchTime < 200);
```

Debugging

Delphi Plugin

Debug Output

```pascal
{$IFDEF DEBUG}
OutputDebugString(PChar('IDESwitcher: ' + Message));
{$ENDIF}
```

Common Issues

- **OTA not available**: Check BorlandIDEServices
- **Pipe creation fails**: Check permissions
- **File not found**: Verify path resolution

VS Code Extension

Debug Console

```typescript
console.log('IDE Switcher:', message);
```

Developer Tools

- F1 → "Developer: Toggle Developer Tools"
- Check Console for errors
- Network tab won't show pipes (local IPC)

Common Issues

- **Pipe connection refused**: Server not running
- **Position off**: Check 0-based vs 1-based indexing
- **File not opening**: Check workspace relative paths

Future Enhancements

Planned Features

1. **State Preservation**
   - Multiple open files
   - Bookmarks
   - Breakpoints
   - Folded regions
   - Selection ranges
2. **Configuration UI**
   - Custom hotkeys
   - Pipe names
   - Timeout settings
   - Auto-save options
3. **Extended Protocol**

```json
{
  "action": "switch",
  "files": [{
    "path": "...",
    "line": 10,
    "column": 5,
    "isActive": true
  }],
  "bookmarks": [...],
  "breakpoints": [...],
  "selection": {
    "start": {"line": 10, "column": 5},
    "end": {"line": 10, "column": 15}
  }
}
```

Cross-Platform Considerations

Linux/Mac Support

- Replace named pipes with Unix domain sockets
- Use different window management APIs
- Adjust file path handling

Network Support

- TCP/IP option for remote development
- SSH tunnel support
- Security considerations for network mode

Code Maintenance

Version Management

- Semantic versioning (major.minor.patch)
- Backwards compatibility for protocol
- Graceful degradation for missing features

Contributing Guidelines

1. Follow existing code style
2. Add tests for new features
3. Update documentation
4. Test on multiple IDE versions
5. Consider backwards compatibility