Imp-to-C Translator: Monitor Backtrace and Event (Signal) Mechanism — Design Notes

Status: Planning document / summary of design discussion
Scope: `monitor.c` / `monitor.h` runtime support library for the Imp77-to-C translator
Purpose: Project documentation for planning next steps; a self-contained summary that can be given to an AI assistant if AI-assisted coding is used in future.

1. Project Goals and Constraints

These constraints were established during the design discussion and should guide all decisions recorded here:

1. **GCC-only.** The translator already relies heavily on GCC extensions (notably nested procedure definitions, which Clang does not support even in GCC-emulation mode). No need to be portable to other C compilers.
2. **Multi-target.** A design goal is that the same runtime code works across multiple GCC target architectures. Avoid per-architecture conditional code (`#ifdef __x86_64__` etc.) wherever possible.
3. **Low runtime-check overhead.** Approximating the behavior of the historical Imp compilers of the 1970s/80s, translated programs should run *by default* with all runtime checks enabled, so checks must be as cheap as possible.
4. **No user environment changes.** Users must not have to set environment variables or change system settings (e.g. enabling core dumps) for the diagnostics to work.
5. **Durable diagnostic output.** The historical environment relied on teletypes / EMAS session logging to capture the backtrace. Modern systems have neither by default, so the monitor should write a copy of its report to a file in addition to the screen.
6. **Reference implementation available.** A mature Imp77 compiler and runtime under VAX/VMS emulation is available for testing edge-case semantics against the historical implementation.

2. How the Current Backtrace Mechanism Works

The runtime is built around a hand-maintained chain of procedure activations, combined with libbacktrace for source-line resolution.

2.1 The Imp-level frame chain

Each translated Imp procedure (and the top-level `%begin/%endofprogram` block, which is translated as an immediately-called anonymous procedure, i.e. C `main`) gets a `FrameContext`:

```
typedef struct FrameContext {
  const char *fn_name;          /* Imp procedure name */
  void (*print_locals)(void);   /* nested function that prints this frame's locals */
  uintptr_t pc_addr;           /* label address inside the procedure, for DWARF looku
  struct FrameContext *parent;  /* dynamic parent activation */
} FrameContext;
```

The translator inserts, at the head of each procedure body:

```
INSERT_BEFORE(name)     /* declares __label__ __trace_loc, a nested backtrace functio
                           builds a FrameContext pointing at the current dynamic pare
                           and sets current_stream_frame = &frame */
```

and at the tail (and at every return point, via the `RETURN` / `RETURN_WITH_RESULT` macros):

```
INSERT_AFTER(name, fmt, ...)  /* restores current_stream_frame = frame.parent and
                                 defines the nested function that prints locals */
```

The `pc_addr` field deliberately takes the address of a *label inside* the procedure rather than the address of the nested function: nested procedure declarations make the apparent function address live in a stack thunk, whereas a label address is a real code address that resolves correctly through DWARF when compiled with `-g`.

`current_stream_frame` is a global pointer to the innermost live activation. `%begin/%endofprogram` is reported as such by special-casing the C function name `main`.

2.2 Printing the backtrace

`_imp_monitor_internal(uintptr_t caller_pc)` walks the frame chain from innermost to outermost. For each frame it:

1. Resolves a source line via libbacktrace (`backtrace_pcinfo`), first using the faulting/monitor PC for the innermost frame, then falling back to each frame's stored `pc_addr`.
2. Prints the offending source line itself by re-reading the source file (`print_source_line_preview`) for the innermost frame.
3. Prints `ENTERED FROM LINE n OF <name>` and calls the frame's `print_locals` nested function.

The `imp_line_callback` used with `backtrace_pcinfo` filters by the frame's `fn_name` so the DWARF walk stops at the correct enclosing scope.

2.3 Entry points into the monitor

Entry point	Trigger	Current behavior
`_imp_monitor()` macro (Imp `%monitor`)	Explicit Imp statement	Prints backtrace, **computation continues** (non-fatal). Uses a local label address as the innermost PC.
`_imp_report_error` family	Unassigned-variable / unassigned-pointer / null-pointer checks (`_ui`, `_ul`, `_up`, `_uf`, `_ud` macros)	Reports via `addr2line` + backtrace, exits.
`_imp_signal_handler`	SIGFPE, SIGSEGV, SIGBUS, SIGILL, SIGTRAP, SIGABRT	Decodes `si_code`, resolves faulting PC, prints backtrace, exits.
`__ubsan_on_report`	GCC UBSan runtime report hook	Prints backtrace, exits.
`_imp_report_error` via `abort()`	Test 8: explicit Imp error	SIGABRT path.

2.4 Unassigned-value detection

Imp77 initializes unassigned variables to a distinctive bit pattern (historically `0x80`-based, e.g. `0x8080808080808080`). The runtime auto-detects the pattern at startup (`detect_unassigned_pattern`, examining an uninitialized stack value) and the check macros (`_ui`, `_up`, `_uf`, etc.) compare values against it before use, reporting `UNASSIGNED VARIABLE` etc. through the same backtrace machinery. The check macros use GCC statement-expressions with a local label address so the reported position anchors to the exact use site.

2.5 Current build command

```
cc -g -O0 -DDBMAIN -o monitor monitor.c \
    -ftrapv \
    -Wno-uninitialized -Wno-maybe-uninitialized -Wno-discarded-qualifiers \
    -fsanitize=undefined \
    -fsanitize=float-divide-by-zero \
    -fsanitize-undefined-trap-on-error \
    -fsanitize=float-cast-overflow \
    -fno-sanitize-recover=all \
    -fstack-protector-all \
    -lm -lbacktrace
```

3. The Central Problem: Trap Mode Destroys the Diagnosis

3.1 What happens today

`-fsanitize-undefined-trap-on-error` tells GCC to replace each sanitizer reporting call with `__builtin_trap()` instead of calling the libubsan reporting routine (see GCC Instrumentation Options). On x86 this compiles to `ud2`, an operand-free invalid-opcode instruction (see Intel SDM, UD2). The consequences:

- The check fires **before** the faulting operation executes (e.g. before the division), so the hardware never raises the SIGFPE the historical Imp compilers produced — the process takes SIGILL instead.
- `ud2` has **no operands and no reason code**. Nothing in the signal information identifies which sanitizer condition failed. The kernel reports an invalid-instruction exception; it does not know whether the trap was inserted for a shift-count violation, a divide-by-zero, an integer overflow, or anything else.
- `siginfo_t.si_code` describes the signal's machine/kernel classification, not the UBSan check. Decoding `si_code` (as `get_sigill_reason` does) can never recover the reason.
- Disassembling around the trap can help a human investigate one binary, but it is not a dependable or portable reporting interface, and it conflicts with the multi-target constraint.

Conclusion: with trap mode, distinguishing "shift too large" from "divide by zero" is not possible in a portable way. (Note: Clang on some targets encodes a handler-category operand in the trap instruction, but this is Clang-specific and irrelevant to this GCC-only project.)

3.2 Recommendation: drop trap mode, keep fail-fast

Remove `-fsanitize-undefined-trap-on-error` and keep `-fno-sanitize-recover=all`. The latter already gives the desired historical behavior — the first detected runtime error terminates execution — but allows the sanitizer runtime to print its diagnostic first. UBSan can then distinguish and name: division by zero, out-of-range shift exponent, invalid shift base, float-to-integer conversion overflow, etc.

Also drop `-ftrapv` from the default build: UBSan's signed-integer-overflow check already covers that ground (including `INT_MIN / -1`), and removing it keeps all arithmetic diagnostics on the single UBSan reporting path instead of mixing trapping mechanisms.

Recommended build:

```
cc -g -O0 -DDBMAIN -o monitor monitor.c \
    -Wno-uninitialized -Wno-maybe-uninitialized -Wno-discarded-qualifiers \
    -fsanitize=undefined \
    -fsanitize=float-divide-by-zero \
    -fsanitize=float-cast-overflow \
    -fno-sanitize-recover=all \
    -fstack-protector-all \
    -lm -lbacktrace
```

Notes:

- `-fsanitize=float-divide-by-zero` and `-fsanitize=float-cast-overflow` are **not** included in `-fsanitize=undefined` and should stay explicit.
- The two floating-point options need individual selection anyway; beyond that, consider trimming the `-fsanitize=undefined` group to only the checks that map onto the Imp runtime-error model, if measurement shows the overhead matters.

- Trap mode remains available as an *optional* minimal-overhead build profile, with the documented trade-off that it loses the specific diagnosis. Measure representative translated programs on each target family before assuming which profile is the right default.
- Check instrumentation cost on the actual targets before committing; the goal is "checks on by default with acceptable overhead," which is an empirical question per target.

4. Issues in the Current Monitor Code to Fix

These are concrete defects or risks identified in the current `monitor.c`:

1. `__ubsan_on_report` **exits too early and uses the wrong PC.**
   - Exiting inside the hook can prevent the UBSan runtime from completing the report that contains the diagnosis. Instead: print the Imp-level backtrace from the hook, then **return**, and let UBSan print its diagnostic and terminate (with `-fno-sanitize-recover=all`).
   - `__builtin_return_address(0)` inside the hook is the caller *inside the sanitizer runtime*, not the failing expression. Either (a) use the report's own filename/line for the innermost location and keep the Imp chain for enclosing frames, or (b) walk the physical stack from the hook and match frames against `current_stream_frame->fn_name`. Investigate `__ubsan_get_current_report_data` (present in libubsan sources) only if you pin and test the GCC/libubsan versions in use — do not treat it as a stable interface.
2. `si_addr` **is used as the faulting PC.** For SIGSEGV/SIGBUS, `si_addr` is the *accessed address*, not the instruction pointer; for SIGFPE it is often 0 or the faulting instruction depending on platform. Use the saved instruction pointer from `ucontext` (the handler already receives it) when the interrupted PC is needed. This needs care per target; where possible prefer the sanitizer/monitor path over raw signal PCs.
3. **Async-signal-safety.** `_imp_signal_handler` calls `fprintf`, `strsignal`, libbacktrace, and `exit` — none are async-signal-safe. For genuinely fatal hardware faults this is best-effort by design (the process is dying anyway), but be aware that a signal arriving while inside one of those functions can deadlock or crash inside the handler. Keep the fatal-signal path minimal; do the structured reporting from the UBSan hook (an ordinary runtime callback) wherever possible.
4. `find_user_pc_callback` **filter is too loose.** It accepts "any named function not starting with `_imp_`", which does not exclude sanitizer-runtime or libc internals. Consider matching against the known innermost Imp frame name instead.
5. `feenableexcept(FE_INVALID | FE_DIVBYZERO)` **conflicts with UBSan float checks.** GCC explicitly warns that `-fsanitize=float-cast-overflow` does not work well with `FE_INVALID` enabled (see GCC Instrumentation Options). Disable the `feenableexcept` call while validating the UBSan reporting path, and decide separately which floating-point conditions should be hardware traps (historical Imp behavior) versus sanitizer reports.
6. `_imp_report_error` **uses** `popen("addr2line ...")` — fragile (PATH-dependent, shell-dependent, slow, unsafe from a signal path). libbacktrace already provides the same

information; unify on it.

7. **Durable output (new feature).** Add a log-file copy of every monitor report:
    - Predictable default path or command-line option, not an environment variable (constraint 4).
    - Same formatted content to both `stderr` and the file; include executable name, PID, timestamp, error category, source location, and the Imp frame chain with locals.
    - Flush both streams before terminating; handle log-open/write failure without suppressing the terminal diagnostic.
    - Decide explicitly: per-run file (PID/timestamp in name) vs. overwrite "last error" file.

5. Imp Event Mechanism (`%signal` / `%on %event`) — Design

5.1 Required semantics (from the historical language)

- An Imp event is raised by the statement `%signal n,n,n` (three numbers) or by a hardware/runtime error that is trappable.
- An event handler is established by `%on %event <list of event numbers> %start; ...; %finish` inside a procedure (or `%begin`/`%end` block).
- Handlers are **dynamically scoped**: they are removed from the handling chain when their procedure exits.
- An event unwinds the stack to the innermost active handler that accepts its event number; computation continues **from inside that block.** Control is never returned to the raising location.
- A handler can re-raise the event, passing it further up the chain to an earlier (outer) handler.
- If a handler completes without re-raising, the original event trapping of that block is **reinstated.**
- If no handler accepts the event, the diagnostic backtrace is invoked starting from the location where the signal was raised (then terminate).
- **At most one** `%on %event` **block per procedure activation** (including `%begin`/`%end` blocks, which are translated as anonymous procedures). A second declaration, if the compiler allows it, completely replaces the first. One block takes a list of all events of interest; the handler distinguishes them by inspecting fields of the event record: `EVENT_EVENT`, `EVENT_INFO`, `EVENT_EXTRA` (the three `%signal` numbers).
- If a handler does not request an event, it is bypassed as the event passes up the procedure call chain.

5.2 Recommended architecture: Imp events are NOT Unix signals

Do not use `SIGUSR1`/`SIGUSR2` as the transport for Imp events. An Imp `%signal` is a synchronous language-level control transfer; a POSIX signal is an asynchronous OS notification with masks, delivery rules, and restricted handler behavior. Emulating one with the other complicates both and gains nothing. Use a direct call into the Imp runtime dispatcher instead.

The architecture has four layers:

1. **Imp `%signal n`:** ordinary synchronous runtime dispatch to the nearest accepting handler (see 5.3).
2. **`%monitor`:** synchronous, non-unwinding backtrace; already implemented; unchanged.
3. **Fatal runtime faults:** capture minimal fault information, best-effort diagnostic, terminate.
4. **Recoverable hardware events (optional, later):** translate selected Linux signals into Imp events only where recovery is credible and tested per target. The historical compilers could catch e.g. divide-by-zero as events; a few error types were untrappable for implementation reasons. Keep an explicit policy table: which fault categories map to trappable Imp event numbers, which are always fatal, which are implementation-dependent. Candidates:
    - Integer divide-by-zero: potentially trappable, but only if the target gives a reliable way to identify the faulting operation and define the state the handler sees.
    - UBSan-detected undefined behavior: prefer the reporting runtime (see section 3) — its report identifies the failed check; a trap does not.
    - Memory-protection faults, stack corruption, process-integrity failures: diagnostic + terminate; not safely resumable.

5.3 Event dispatch model

Search the **same dynamic parent chain** the monitor uses, from the raising activation outward:

```
Start at the activation where the event is raised.
For each activation, innermost outward:
    If it has a handler accepting EVENT_EVENT:
        select it; stop searching.
    Otherwise continue to its parent.
If none accepts:
    print the diagnostic backtrace at the raising location; terminate.
```

`EVENT_INFO` and `EVENT_EXTRA` are payload delivered to the selected handler, not additional selection criteria.

Example: A accepts {1,3}; B accepts {2}; C accepts {4,5}; C raises event 3 → C and B are bypassed; A receives the record (3, info, extra).

5.4 Handler state: one slot per procedure activation

Because each activation has at most one `%on %event` block, the natural model is a **handler slot owned by the activation** (either embedded in the frame or referenced from it):

```
typedef struct ImpEvent {
    int event;        /* EVENT_EVENT */
    int info;         /* EVENT_INFO  */
    int extra;        /* EVENT_EXTRA */
    /* plus origin information for diagnostics: line number, PC */
} ImpEvent;

typedef struct ImpHandler {
    sigjmp_buf landing;          /* transfer destination for this activation */
    const int *event_numbers;    /* accepted event numbers */
    size_t event_count;
    FrameContext *frame_at_install;  /* activation this handler belongs to */
    ImpEvent event;              /* filled before transfer */
    volatile sig_atomic_t eligible; /* cleared while this handler's body runs */
} ImpHandler;
```

The translated code for an `%on %event ... %start ... %finish` block establishes its landing point with `sigsetjmp` and registers the handler before the body runs:

```
ImpHandler handler = {0};
handler.previous_or_frame_setup(...);
handler.event_numbers = events; handler.event_count = n;
handler.frame_at_install = current_stream_frame;

if (sigsetjmp(handler.landing, 1) == 0) {
    handler.eligible = 1;
    /* ... translated %start body ... */
    /* normal fall-through at %finish: handler becomes eligible again (it already is)
} else {
    /* handler.landing was jumped to: handler.event is the delivered event */
    /* translated event-handling body */
}
```

A second `%on %event` declaration in the same activation replaces the slot's contents.

5.5 Handler eligibility, re-raise, and reinstatement

The subtle part of the semantics:

- When activation A catches an event, **A's own handler becomes temporarily ineligible** while its body executes. A re-raise from within the body must search from A's *parent* outward, not re-select A.
- While A's handler body runs, handlers installed by procedures **called from the body** participate normally. The suppression is activation-local, not a global "current handler position" pointer.

- If the body completes normally (through `%finish`), A's handler is reinstated.
- **Reinstatement must also cover jumps out of the handler body.** Historically programmers put only dispatch code inside handler blocks (an unstated convention, never codified), but syntactically a `goto` from inside the `%on %event` block to elsewhere in the same procedure is allowed, and this case may never have been tested on any implementation. Proposed semantic interpretation (not yet verified against the historical compiler): re-enable eligibility whenever execution leaves the handler body normally, including a jump to another part of the same activation.

Exit cases to handle in the translator:

Exit from handler body	Required action
Fall through `%finish`	Re-enable this activation's handler.
Jump outside the block, same activation	Re-enable before transferring (proposed; verify against VAX/VMS).
Return from the procedure	Ordinary procedure-exit bookkeeping removes the activation.
Re-raise an event	Search outward while this handler stays ineligible.
Call another procedure	Keep this handler ineligible during the call; callee's handlers work normally.

5.6 Interaction with the frame chain — critical invariant

`FrameContext` objects are automatic variables linked through `current_stream_frame`, and normal procedure exits restore the parent pointer. A `siglongjmp` over one or more procedures **bypasses those exits**, leaving `current_stream_frame` pointing at dead stack frames. Before transferring to the selected handler:

- Reset `current_stream_frame` to the handler's `frame_at_install` (or the frame representing the event-block scope).
- Apply the same restoration to every other runtime-managed dynamic state (handler eligibility, any cleanup records).

Without this, a subsequent `%monitor` or an unhandled event walks stale pointers.

5.7 `siglongjmp` cautions

- For Imp's own `%signal`, the dispatcher performs the nonlocal transfer directly — no POSIX signal is involved, so this is just ordinary `sigsetjmp`/`siglongjmp` discipline.
- For a Linux signal handler, POSIX warns against `siglongjmp` when the signal interrupted a non-async-signal-safe function and the resumed code later calls such functions (see POSIX `sigsetjmp` rationale). The monitor uses stdio, file operations, and libbacktrace, so hardware-fault-to-event bridging (section 5.2 layer 4) must be added conservatively, per fault class, per target.
- Never leave a pointer to an automatic `ImpHandler` installed after its C scope or function has ended (GCC nested functions and their frame references must not outlive their containing activation).

5.8 Unhandled events and diagnostics

Decide explicitly what the final backtrace reports when an event (possibly after re-raises) goes unhandled: the original raise, the last re-raise, or both. This determines what origin information `ImpEvent` must retain.

6. Validation Plan

6.1 VAX/VMS reference testing (edge cases in event semantics)

The mature Imp77 compiler under VAX/VMS emulation is the reference for the ambiguous cases. Keep each case in a separate test program with unmistakable output (an outer handler that prints) to distinguish "inner handler reinstated" from "event escaped outward":

Handler action	Subsequent test
Falls through `%finish`	Raise the same event again in the procedure body.
Jumps to a label after `%finish`	Raise the same event at the destination.
Jumps to a label before the event block	Reach another raise without re-executing the handler declaration.
Jumps between labels inside the handler	Raise the same event while still inside it.
Calls another procedure	Raise an event there, with and without a matching handler in the callee.

Also add to the validation suite:

- Compiler test: does a second `%on %event` block in one procedure compile, and does it replace the first? (Currently untested.)
- Handler calls a procedure that catches a new event.
- Handler calls a procedure whose event must pass outward beyond the currently executing handler.
- Handler jumps to another label, then a second event is caught by the reinstated handler.
- Recursive call creates another activation of the same procedure; its handler operates independently of the suspended handler in its caller.
- Handler re-raises an event that ultimately remains unhandled (diagnostic policy, 5.8).

6.2 Documentation vs. implementation decisions

Distinguish, in notes and documentation, three categories:

1. Language behavior established by the language documentation.
2. Behavior observed in the VAX/VMS implementation.
3. Behavior explicitly supported or restricted by this translator.

If detecting jumps out of a handler body would require disproportionate code-generator work, a precise restriction is acceptable, e.g.:

> An event-handler body may complete through its `%finish`, return from the containing procedure, or raise an event. A jump from inside the handler body to a label outside that body is not supported.

Distinguish "rejected by the compiler" (needs a diagnostic the translator can reliably generate) from "documented as unsupported."

7. Recommended Next Steps (Priority Order)

1. **Switch the default build to non-trap UBSan** (section 3.2): remove `-fsanitize-undefined-trap-on-error` and `-ftrapv`, keep `-fno-sanitize-recover=all`. This immediately answers the original question ("which trap was it?") by restoring the sanitizer's diagnosis.
2. **Rework** `__ubsan_on_report`: print the Imp backtrace from the hook, return instead of exiting, let UBSan print its report and terminate; fix the innermost-PC resolution (report data or frame-name matching rather than `__builtin_return_address`).
3. **Fix** `si_addr`/`ucontext` **PC handling** in the fatal-signal path.
4. **Add the log-file copy** of monitor reports (section 4.7), per the durable-output constraint.
5. **Run the VAX/VMS edge-case matrix** (section 6.1) to pin down handler-reinstatement semantics before implementing the event mechanism.
6. **Implement Imp events** per section 5: dispatcher over the existing frame chain, one handler slot per activation, activation-local eligibility, `sigsetjmp`/`siglongjmp` transfer with frame-chain restoration.
7. **Optionally bridge recoverable hardware faults** to Imp events later, per the policy table, per target.
8. **Defer or document** the jump-out-of-handler restriction if code-generator changes prove disproportionate.

8. References

- GCC instrumentation options (`-fsanitize-undefined-trap-on-error`, `-fno-sanitize-recover`, float-cast-overflow / FE_INVALID interaction): https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
- Intel Architecture Software Developer's Manual, `UD2` instruction: https://www.intel.com/content/www/us/en/content-details/334528
- POSIX `sigsetjmp`/`siglongjmp` (signal-handler interaction cautions): https://pubs.opengroup.org/onlinepubs/9699919799/functions/sigsetjmp.html
- Project source: `monitor.c`, `monitor.h` (Imp-to-C translator runtime, as of this writing)