```
MODULE Dev2CPT;
(**

    project         = "BlackBox"
    organization    = "www.oberon.ch"
    contributors    = "Oberon microsystems"
    version         = "System/Rsrc/About"
    copyright       = "System/Rsrc/About"
    license         = "Docu/BB-License"
    references      = "http://e-collection.library.ethz.ch/eserv/eth:39386/eth-39386-02.pdf"
    changes         = ""
    issues          = ""

**)

    IMPORT Dev2CPM;

    CONST
        MaxIdLen = 256;

    TYPE
        Name* = ARRAY MaxIdLen OF SHORTCHAR;
        String* = POINTER TO ARRAY OF SHORTCHAR;
        Const* = POINTER TO ConstDesc;
        Object* = POINTER TO ObjDesc;
        Struct* = POINTER TO StrDesc;
        Node* = POINTER TO NodeDesc;
        ConstExt* = String;
        LinkList* = POINTER TO LinkDesc;

        ConstDesc* = RECORD
            ext*: ConstExt;    (* string or code for code proc (longstring in utf8) *)
            intval*: INTEGER;    (* constant value or adr, proc par size, text position or least case label *)
            intval2*: INTEGER;    (* string length (#char, incl 0X), proc var size or larger case label *)
            setval*: SET;    (* constant value, procedure body present or "ELSE" present in case *)
            realval*: REAL;    (* real or longreal constant value *)
            link*: Const    (* chain of constants present in obj file *)
        END ;

        LinkDesc* = RECORD
            offset*, linkadr*: INTEGER;
            next*: LinkList;
        END;

        ObjDesc* = RECORD
            left*, right*, link*, scope*: Object;
            name*: String;    (* name = null OR name^ # "" *)
            leaf*: BOOLEAN;
            sysflag*: BYTE;
            mode*, mnolev*: BYTE;    (* mnolev < 0 -> mno = -mnolev *)
            vis*: BYTE;    (* internal, external, externalR, inPar, outPar *)
            history*: BYTE;    (* relevant if name # "" *)
            used*, fpdone*: BOOLEAN;
            fprint*: INTEGER;
            typ*: Struct;    (* actual type, changed in with statements *)
            ptyp*: Struct;    (* original type if typ is changed *)
            conval*: Const;
            adr*, num*: INTEGER;    (* mthno *)
            links*: LinkList;
            nlink*: Object;    (* link for name list, declaration order for methods, library link for imp obj *)
```

```
		library*, entry*: String;	(* library name, entry name *)
		modifiers*: POINTER TO ARRAY OF String;	(* additional interface strings *)
		linkadr*: INTEGER;	(* used in ofront *)
		red: BOOLEAN;
	END ;

	StrDesc* = RECORD
		form*, comp*, mno*, extlev*: BYTE;
		ref*, sysflag*: SHORTINT;
		n*, size*, align*, txtpos*: INTEGER;	(* align is alignment for records and len offset for dynarrs *)
		untagged*, allocated*, pbused*, pvused*, exp*, fpdone, idfpdone: BOOLEAN;
		attribute*: BYTE;
		idfp, pbfp*, pvfp*:INTEGER;
		BaseTyp*: Struct;
		link*, strobj*: Object;
		ext*: ConstExt	(* id string for interface records *)
	END ;

	NodeDesc* = RECORD
		left*, right*, link*: Node;
		class*, subcl*, hint*: BYTE;
		readonly*: BOOLEAN;
		typ*: Struct;
		obj*: Object;
		conval*: Const
	END ;

CONST
	maxImps = 127;	(* must be <= MAX(SHORTINT) *)
	maxStruct = Dev2CPM.MaxStruct;	(* must be < MAX(INTEGER) DIV 2 *)
	FirstRef = 32;
	FirstRef0 = 16;	(* correction for version 0 *)
	actVersion = 1;

VAR
	topScope*: Object;
	undftyp*, bytetyp*, booltyp*, char8typ*, int8typ*, int16typ*, int32typ*,
	real32typ*, real64typ*, settyp*, string8typ*, niltyp*, notyp*, sysptrtyp*,
	anytyp*, anyptrtyp*, char16typ*, string16typ*, int64typ*,
	restyp*, iunktyp*, punktyp*, guidtyp*,
	intrealtyp*, lreal64typ*, lint64typ*, lchar16typ*: Struct;
	nofGmod*: BYTE;	(*nof imports*)
	GlbMod*: ARRAY maxImps OF Object;	(* .right = first object, .name = module import name (not alias) *)
	SelfName*: Name;	(* name of module being compiled *)
	SYSimported*: BOOLEAN;
	processor*, impProc*: SHORTINT;
	libName*: Name;	(* library alias of module being compiled *)
	null*: String;	(* "" *)

CONST
	(* object modes *)
	Var = 1; VarPar = 2; Con = 3; Fld = 4; Typ = 5; LProc = 6; XProc = 7;
	SProc = 8; CProc = 9; IProc = 10; Mod = 11; Head = 12; TProc = 13; Attr = 20;

	(* structure forms *)
	Undef = 0; Byte = 1; Bool = 2; Char8 = 3; Int8 = 4; Int16 = 5; Int32 = 6;
	Real32 = 7; Real64 = 8; Set = 9; String8 = 10; NilTyp = 11; NoTyp = 12;
	Pointer = 13; ProcTyp = 14; Comp = 15;
	AnyPtr = 14; AnyRec = 15;	(* sym file only *)
	Char16 = 16; String16 = 17; Int64 = 18;
```

```
		Res = 20; IUnk = 21; PUnk = 22; Guid = 23;

		(* composite structure forms *)
		Basic = 1; Array = 2; DynArr = 3; Record = 4;

		(*function number*)
		assign = 0;
		haltfn = 0; newfn = 1; absfn = 2; capfn = 3; ordfn = 4;
		entierfn = 5; oddfn = 6; minfn = 7; maxfn = 8; chrfn = 9;
		shortfn = 10; longfn = 11; sizefn = 12; incfn = 13; decfn = 14;
		inclfn = 15; exclfn = 16; lenfn = 17; copyfn = 18; ashfn = 19; assertfn = 32;
		lchrfn = 33; lentierfcn = 34; typfn = 36; bitsfn = 37; bytesfn = 38;

		(*SYSTEM function number*)
		adrfn = 20; ccfn = 21; lshfn = 22; rotfn = 23;
		getfn = 24; putfn = 25; getrfn = 26; putrfn = 27;
		bitfn = 28; valfn = 29; sysnewfn = 30; movefn = 31;
		thisrecfn = 45; thisarrfn = 46;

		(* COM function number *)
		validfn = 40; iidfn = 41; queryfn = 42;

		(* attribute flags (attr.adr, struct.attribute, proc.conval.setval) *)
		newAttr = 16; absAttr = 17; limAttr = 18; empAttr = 19; extAttr = 20;

		(* procedure flags (conval.setval) *)
		isHidden = 29;

		(* module visibility of objects *)
		internal = 0; external = 1; externalR = 2; inPar = 3; outPar = 4;

		(* history of imported objects *)
		inserted = 0; same = 1; pbmodified = 2; pvmodified = 3; removed = 4; inconsistent = 5;

		(* sysflags *)
		inBit = 2; outBit = 4; interface = 10;

		(* symbol file items *)
		Smname = 16; Send = 18; Stype = 19; Salias = 20; Svar = 21; Srvar = 22;
		Svalpar = 23; Svarpar = 24; Sfld = 25; Srfld = 26; Shdptr = 27; Shdpro = 28; Stpro = 29; Shdtpro = 30;
		Sxpro = 31; Sipro = 32; Scpro = 33; Sstruct = 34; Ssys = 35; Sptr = 36; Sarr = 37; Sdarr = 38; Srec = 39; Spro = 40;
		Shdutptr = 41; Slib = 42; Sentry = 43; Sinpar = 25; Soutpar = 26;
		Slimrec = 25; Sabsrec = 26; Sextrec = 27; Slimpro = 31; Sabspro = 32; Semppro = 33; Sextpro = 34; Simpo = 22;

	TYPE
		ImpCtxt = RECORD
			nextTag, reffp: INTEGER;
			nofr, minr, nofm: SHORTINT;
			self: BOOLEAN;
			ref: ARRAY maxStruct OF Struct;
			old: ARRAY maxStruct OF Object;
			pvfp: ARRAY maxStruct OF INTEGER;	(* set only if old # NIL *)
			glbmno: ARRAY maxImps OF BYTE	(* index is local mno *)
		END ;

		ExpCtxt = RECORD
			reffp: INTEGER;
			ref: SHORTINT;
			nofm: BYTE;
			locmno: ARRAY maxImps OF BYTE	(* index is global mno *)
```

```
    END ;

VAR
    universe, syslink, comlink, infinity: Object;
    impCtxt: ImpCtxt;
    expCtxt: ExpCtxt;
    nofhdfld: INTEGER;
    sfpresent, symExtended, symNew: BOOLEAN;
    version: INTEGER;
    symChanges: INTEGER;
    portable: BOOLEAN;
    depth: INTEGER;


PROCEDURE err(n: SHORTINT);
BEGIN Dev2CPM.err(n)
END err;

PROCEDURE NewConst*(): Const;
    VAR const: Const;
BEGIN NEW(const); RETURN const
END NewConst;

PROCEDURE NewObj*(): Object;
    VAR obj: Object;
BEGIN NEW(obj); obj.name := null; RETURN obj
END NewObj;

PROCEDURE NewStr*(form, comp: BYTE): Struct;
    VAR typ: Struct;
BEGIN NEW(typ); typ.form := form; typ.comp := comp; typ.ref := maxStruct; (* ref >= maxStruct: not exported yet *)
    typ.txtpos := Dev2CPM.errpos; typ.size := -1; typ.BaseTyp := undftyp; RETURN typ
END NewStr;

PROCEDURE NewNode*(class: BYTE): Node;
    VAR node: Node;
BEGIN
    NEW(node); node.class := class; RETURN node
END NewNode;

PROCEDURE NewName* (IN name: ARRAY OF SHORTCHAR): String;
    VAR i: INTEGER; p: String;
BEGIN
    i := LEN(name$);
    IF i > 0 THEN NEW(p, i + 1); p^ := name$; RETURN p
    ELSE RETURN null
    END
END NewName;

PROCEDURE OpenScope*(level: BYTE; owner: Object);
    VAR head: Object;
BEGIN head := NewObj();
    head.mode := Head; head.mnolev := level; head.link := owner;
    IF owner # NIL THEN owner.scope := head END ;
    head.left := topScope; head.right := NIL; head.scope := NIL; topScope := head
END OpenScope;

PROCEDURE CloseScope*;
BEGIN topScope := topScope.left
END CloseScope;
```

```
PROCEDURE Init*(opt: SET);
BEGIN
    topScope := universe; OpenScope(0, NIL); SYSimported := FALSE;
    GlbMod[0] := topScope; nofGmod := 1;
    sfpresent := TRUE;     (* !!! *)
    symChanges := 0;
    infinity.conval.intval := Dev2CPM.ConstNotAlloc;
    depth := 0
END Init;

PROCEDURE Open* (IN name: Name);
BEGIN
    SelfName := name$; topScope.name := NewName(name);
END Open;

PROCEDURE Close*;
    VAR i: INTEGER;
BEGIN    (* garbage collection *)
    CloseScope;
    i := 0; WHILE i < maxImps DO GlbMod[i] := NIL; INC(i) END ;
    i := FirstRef; WHILE i < maxStruct DO impCtxt.ref[i] := NIL; impCtxt.old[i] := NIL; INC(i) END
END Close;

PROCEDURE SameType* (x, y: Struct): BOOLEAN;
BEGIN
    RETURN (x = y) OR (x.form = y.form) & ~(x.form IN {Pointer, ProcTyp, Comp}) OR (x = undftyp) OR (y = undftyp)
END SameType;

PROCEDURE EqualType* (x, y: Struct): BOOLEAN;
    VAR xp, yp: Object; n: INTEGER;
BEGIN
    n := 0;
    WHILE (n < 100) & (x # y)
        & (((x.comp = DynArr) & (y.comp = DynArr) & (x.sysflag = y.sysflag))
            OR ((x.form = Pointer) & (y.form = Pointer))
            OR ((x.form = ProcTyp) & (y.form = ProcTyp))) DO
        IF x.form = ProcTyp THEN
            IF x.sysflag # y.sysflag THEN RETURN FALSE END;
            xp := x.link; yp := y.link;
            INC(depth);
            WHILE (xp # NIL) & (yp # NIL) & (xp.mode = yp.mode) & (xp.sysflag = yp.sysflag)
                    & (xp.vis = yp.vis) & (depth < 100) & EqualType(xp.typ, yp.typ) DO
                xp := xp.link; yp := yp.link
            END;
            DEC(depth);
            IF (xp # NIL) OR (yp # NIL) THEN RETURN FALSE END
        END;
        x := x.BaseTyp; y := y.BaseTyp; INC(n)
    END;
    RETURN SameType(x, y)
END EqualType;

PROCEDURE Extends* (x, y: Struct): BOOLEAN;
BEGIN
    IF (x.form = Pointer) & (y.form = Pointer) THEN x := x.BaseTyp; y := y.BaseTyp END;
    IF (x.comp = Record) & (y.comp = Record) THEN
        IF (y = anytyp) & ~x.untagged THEN RETURN TRUE END;
        WHILE (x # NIL) & (x # undftyp) & (x # y) DO x := x.BaseTyp END
    END;
    RETURN (x # NIL) & EqualType(x, y)
```

```
	END Extends;

	PROCEDURE Includes* (xform, yform: INTEGER): BOOLEAN;
	BEGIN
		CASE xform OF
		| Char16: RETURN yform IN {Char8, Char16, Int8}
		| Int16: RETURN yform IN {Char8, Int8, Int16}
		| Int32: RETURN yform IN {Char8, Char16, Int8, Int16, Int32}
		| Int64: RETURN yform IN {Char8, Char16, Int8, Int16, Int32, Int64}
		| Real32: RETURN yform IN {Char8, Char16, Int8, Int16, Int32, Int64, Real32}
		| Real64: RETURN yform IN {Char8, Char16, Int8, Int16, Int32, Int64, Real32, Real64}
		| String16: RETURN yform IN {String8, String16}
		ELSE RETURN xform = yform
		END
	END Includes;

	PROCEDURE FindImport*(IN name: Name; mod: Object; VAR res: Object);
		VAR obj: Object; (* i: INTEGER; n: Name; *)
	BEGIN obj := mod.scope.right;
		LOOP
			IF obj = NIL THEN EXIT END ;
			IF name < obj.name^ THEN obj := obj.left
			ELSIF name > obj.name^ THEN obj := obj.right
			ELSE (*found*)
				IF (obj.mode = Typ) & (obj.vis = internal) THEN obj := NIL
				ELSE obj.used := TRUE
				END ;
				EXIT
			END
		END ;
		res := obj;
(*	bh: checks usage of non Unicode WinApi functions and types
		IF (res # NIL) & (mod.scope.library # NIL)
				& ~(Dev2CPM.interface IN Dev2CPM.options)
				& (SelfName # "Kernel") & (SelfName # "HostPorts") THEN
			n := name + "W";
			FindImport(n, mod, obj);
			IF obj # NIL THEN
				Dev2CPM.err(733)
			ELSE
				i := LEN(name$);
				IF name[i - 1] = "A" THEN
					n[i - 1] := "W"; n[i] := 0X;
					FindImport(n, mod, obj);
					IF obj # NIL THEN
						Dev2CPM.err(734)
					END
				END
			END
		END;
*)
	END FindImport;

	PROCEDURE Find*(IN name: Name; VAR res: Object);
		VAR obj, head: Object;
	BEGIN head := topScope;
		LOOP obj := head.right;
			LOOP
				IF obj = NIL THEN EXIT END ;
				IF name < obj.name^ THEN obj := obj.left
				ELSIF name > obj.name^ THEN obj := obj.right
				ELSE (* found, obj.used not set for local objects *) EXIT
				END
			END ;
			IF obj # NIL THEN EXIT END ;
			head := head.left;
			IF head = NIL THEN EXIT END
		END ;
		res := obj
	END Find;

	PROCEDURE FindFld (IN name: ARRAY OF SHORTCHAR; typ: Struct; VAR res: Object);
		VAR obj: Object;
	BEGIN
		WHILE (typ # NIL) & (typ # undftyp) DO obj := typ.link;
			WHILE obj # NIL DO
				IF name < obj.name^ THEN obj := obj.left
				ELSIF name > obj.name^ THEN obj := obj.right
				ELSE (*found*) res := obj; RETURN
				END
			END ;
			typ := typ.BaseTyp
		END;
		res := NIL
	END FindFld;

	PROCEDURE FindField* (IN name: ARRAY OF SHORTCHAR; typ: Struct; VAR res: Object);
	BEGIN
		FindFld(name, typ, res);
		IF (res = NIL) & ~typ.untagged THEN FindFld(name, anytyp, res) END
	END FindField;

	PROCEDURE FindBaseField* (IN name: ARRAY OF SHORTCHAR; typ: Struct; VAR res: Object);
	BEGIN
		FindFld(name, typ.BaseTyp, res);
		IF (res = NIL) & ~typ.untagged THEN FindFld(name, anytyp, res) END
	END FindBaseField;

(*
	PROCEDURE Rotated (y: Object; name: String): Object;
		VAR c, gc: Object;
	BEGIN
		IF name^ < y.name^ THEN
			c := y.left;
			IF name^ < c.name^ THEN gc := c.left; c.left := gc.right; gc.right := c
			ELSE gc := c.right; c.right := gc.left; gc.left := c
			END;
			y.left := gc
		ELSE
			c := y.right;
			IF name^ < c.name^ THEN gc := c.left; c.left := gc.right; gc.right := c
			ELSE gc := c.right; c.right := gc.left; gc.left := c
			END;
			y.right := gc
		END;
		RETURN gc
	END Rotated;

	PROCEDURE InsertIn (obj, scope: Object; VAR old: Object);
		VAR gg, g, p, x: Object; name, sname: String;
```

```
    BEGIN
        sname := scope.name; scope.name := null;
        gg := scope; g := gg; p := g; x := p.right; name := obj.name;
        WHILE x # NIL DO
            IF (x.left # NIL) & (x.right # NIL) & x.left.red & x.right.red THEN
                x.red := TRUE; x.left.red := FALSE; x.right.red := FALSE;
                IF p.red THEN
                    g.red := TRUE;
                    IF (name^ < g.name^) # (name^ < p.name^) THEN p := Rotated(g, name) END;
                    x := Rotated(gg, name); x.red := FALSE
                END
            END;
            gg := g; g := p; p := x;
            IF name^ < x.name^ THEN x := x.left
            ELSIF name^ > x.name^ THEN x := x.right
            ELSE old := x; scope.right.red := FALSE; scope.name := sname; RETURN
            END
        END;
        x := obj; old := NIL;
        IF name^ < p.name^ THEN p.left := x ELSE p.right := x END;
        x.red := TRUE;
        IF p.red THEN
            g.red := TRUE;
            IF (name^ < g.name^) # (name^ < p.name^) THEN p := Rotated(g, name) END;
            x := Rotated(gg, name);
            x.red := FALSE
        END;
        scope.right.red := FALSE; scope.name := sname
    END InsertIn;
*)
    PROCEDURE InsertIn (obj, scope: Object; VAR old: Object);
        VAR ob0, ob1: Object; left: BOOLEAN; name: String;
    BEGIN
        ASSERT((scope # NIL) & (scope.mode = Head), 100);
        ob0 := scope; ob1 := scope.right; left := FALSE; name := obj.name;
        WHILE ob1 # NIL DO
            IF name^ < ob1.name^ THEN ob0 := ob1; ob1 := ob1.left; left := TRUE
            ELSIF name^ > ob1.name^ THEN ob0 := ob1; ob1 := ob1.right; left := FALSE
            ELSE old := ob1; RETURN
            END
        END;
        IF left THEN ob0.left := obj ELSE ob0.right := obj END ;
        obj.left := NIL; obj.right := NIL; old := NIL
    END InsertIn;

    PROCEDURE Insert* (IN name: Name; VAR obj: Object);
        VAR old: Object;
    BEGIN
        obj := NewObj(); obj.leaf := TRUE;
        obj.name := NewName(name);
        obj.mnolev := topScope.mnolev;
        InsertIn(obj, topScope, old);
        IF old # NIL THEN err(1) END     (*double def*)
    END Insert;

    PROCEDURE InsertThisField (obj: Object; typ: Struct; VAR old: Object);
        VAR ob0, ob1: Object; left: BOOLEAN; name: String;
    BEGIN
        IF typ.link = NIL THEN typ.link := obj
        ELSE
```

```
			ob1 := typ.link; name := obj.name;
			REPEAT
				IF name^ < ob1.name^ THEN ob0 := ob1; ob1 := ob1.left; left := TRUE
				ELSIF name^ > ob1.name^ THEN ob0 := ob1; ob1 := ob1.right; left := FALSE
				ELSE old := ob1; RETURN
				END
			UNTIL ob1 = NIL;
			IF left THEN ob0.left := obj ELSE ob0.right := obj END
		END
	END InsertThisField;

	PROCEDURE InsertField* (IN name: Name; typ: Struct; VAR obj: Object);
		VAR old: Object;
	BEGIN
		obj := NewObj(); obj.leaf := TRUE;
		obj.name := NewName(name);
		InsertThisField(obj, typ, old);
		IF old # NIL THEN err(1) END	(*double def*)
	END InsertField;


(*-------------------------- Fingerprinting --------------------------*)

	PROCEDURE FPrintName(VAR fp: INTEGER; IN name: ARRAY OF SHORTCHAR);
		VAR i: INTEGER; ch: SHORTCHAR;
	BEGIN i := 0;
		REPEAT ch := name[i]; Dev2CPM.FPrint(fp, ORD(ch)); INC(i) UNTIL ch = 0X
	END FPrintName;

	PROCEDURE ^IdFPrint*(typ: Struct);

	PROCEDURE FPrintSign*(VAR fp: INTEGER; result: Struct; par: Object);
	(* depends on assignment compatibility of params only *)
	BEGIN
		IdFPrint(result); Dev2CPM.FPrint(fp, result.idfp);
		WHILE par # NIL DO
			Dev2CPM.FPrint(fp, par.mode); IdFPrint(par.typ);
			Dev2CPM.FPrint(fp, par.typ.idfp);
			IF (par.mode = VarPar) & (par.vis # 0) THEN Dev2CPM.FPrint(fp, par.vis) END;	(* IN / OUT *)
			IF par.sysflag # 0 THEN Dev2CPM.FPrint(fp, par.sysflag) END;
			(* par.name and par.adr not considered *)
			par := par.link
		END
	END FPrintSign;

	PROCEDURE IdFPrint*(typ: Struct);	(* idfp codifies assignment compatibility *)
		VAR btyp: Struct; strobj: Object; idfp: INTEGER; f, c: SHORTINT;
	BEGIN
		IF ~typ.idfpdone THEN
			typ.idfpdone := TRUE;	(* may be recursive, temporary idfp is 0 in that case *)
			idfp := 0; f := typ.form; c := typ.comp; Dev2CPM.FPrint(idfp, f); Dev2CPM.FPrint(idfp, c);
			btyp := typ.BaseTyp; strobj := typ.strobj;
			IF (strobj # NIL) & (strobj.name # null) THEN
				FPrintName(idfp, GlbMod[typ.mno].name^); FPrintName(idfp, strobj.name^)
			END ;
			IF (f = Pointer) OR (c = Record) & (btyp # NIL) OR (c = DynArr) THEN
				IdFPrint(btyp); Dev2CPM.FPrint(idfp, btyp.idfp)
			ELSIF c = Array THEN IdFPrint(btyp); Dev2CPM.FPrint(idfp, btyp.idfp); Dev2CPM.FPrint(idfp, typ.n)
			ELSIF f = ProcTyp THEN FPrintSign(idfp, btyp, typ.link)
			END ;
```

```
        IF typ.sysflag # 0 THEN Dev2CPM.FPrint(idfp, typ.sysflag) END;
        typ.idfp := idfp
    END
END IdFPrint;

PROCEDURE FPrintStr*(typ: Struct);
    VAR f, c: SHORTINT; btyp: Struct; strobj, bstrobj: Object; pbfp, pvfp: INTEGER;

    PROCEDURE ^FPrintFlds(fld: Object; adr: INTEGER; visible: BOOLEAN);

    PROCEDURE FPrintHdFld(typ: Struct; fld: Object; adr: INTEGER);    (* modifies pvfp only *)
        VAR i, j, n: INTEGER; btyp: Struct;
    BEGIN
        IF typ.comp = Record THEN
            IF typ.BaseTyp # NIL THEN FPrintHdFld(typ.BaseTyp, fld, adr) END ;
            FPrintFlds(typ.link, adr, FALSE)
        ELSIF typ.comp = Array THEN btyp := typ.BaseTyp; n := typ.n;
            WHILE btyp.comp = Array DO n := btyp.n * n; btyp := btyp.BaseTyp END ;
            IF (btyp.form = Pointer) OR (btyp.comp = Record) THEN
                j := nofhdfld; FPrintHdFld(btyp, fld, adr);
                IF j # nofhdfld THEN i := 1;
                    WHILE (i < n) (* & (nofhdfld <= Dev2CPM.MaxHdFld) *) DO    (* !!! *)
                        INC(adr, btyp.size); FPrintHdFld(btyp, fld, adr); INC(i)
                    END
                END
            END
        ELSIF Dev2CPM.ExpHdPtrFld &
            ((typ.form = Pointer) & ~typ.untagged OR (fld.name^ = Dev2CPM.HdPtrName)) THEN    (* !!! *)
            Dev2CPM.FPrint(pvfp, Pointer); Dev2CPM.FPrint(pvfp, adr); INC(nofhdfld)
        ELSIF Dev2CPM.ExpHdUtPtrFld &
            ((typ.form = Pointer) & typ.untagged OR (fld.name^ = Dev2CPM.HdUtPtrName)) THEN    (* !!! *)
            Dev2CPM.FPrint(pvfp, Pointer); Dev2CPM.FPrint(pvfp, adr); INC(nofhdfld);
            IF typ.form = Pointer THEN Dev2CPM.FPrint(pvfp, typ.sysflag) ELSE Dev2CPM.FPrint(pvfp, fld.sysflag) END
        ELSIF Dev2CPM.ExpHdProcFld & ((typ.form = ProcTyp) OR (fld.name^ = Dev2CPM.HdProcName)) THEN
            Dev2CPM.FPrint(pvfp, ProcTyp); Dev2CPM.FPrint(pvfp, adr); INC(nofhdfld)
        END
    END FPrintHdFld;

    PROCEDURE FPrintFlds(fld: Object; adr: INTEGER; visible: BOOLEAN);    (* modifies pbfp and pvfp *)
    BEGIN
        WHILE (fld # NIL) & (fld.mode = Fld) DO
            IF (fld.vis # internal) & visible THEN
                Dev2CPM.FPrint(pvfp, fld.vis); FPrintName(pvfp, fld.name); Dev2CPM.FPrint(pvfp, fld.adr);
                Dev2CPM.FPrint(pbfp, fld.vis); FPrintName(pbfp, fld.name); Dev2CPM.FPrint(pbfp, fld.adr);
                FPrintStr(fld.typ); Dev2CPM.FPrint(pbfp, fld.typ.pbfp); Dev2CPM.FPrint(pvfp, fld.typ.pvfp)
            ELSE FPrintHdFld(fld.typ, fld, fld.adr + adr)
            END ;
            fld := fld.link
        END
    END FPrintFlds;

    PROCEDURE FPrintTProcs(obj: Object);    (* modifies pbfp and pvfp *)
        VAR fp: INTEGER;
    BEGIN
        IF obj # NIL THEN
            FPrintTProcs(obj.left);
            IF obj.mode = TProc THEN
                IF obj.vis # internal THEN
                    fp := 0;
                    IF obj.vis = externalR THEN Dev2CPM.FPrint(fp, externalR) END;
                    IF limAttr IN obj.conval.setval THEN Dev2CPM.FPrint(fp, limAttr)
                    ELSIF absAttr IN obj.conval.setval THEN Dev2CPM.FPrint(fp, absAttr)
                    ELSIF empAttr IN obj.conval.setval THEN Dev2CPM.FPrint(fp, empAttr)
                    ELSIF extAttr IN obj.conval.setval THEN Dev2CPM.FPrint(fp, extAttr)
                    END;
                    Dev2CPM.FPrint(fp, TProc); Dev2CPM.FPrint(fp, obj.num);
                    FPrintSign(fp, obj.typ, obj.link); FPrintName(fp, obj.name);
                    IF obj.entry # NIL THEN FPrintName(fp, obj.entry) END;
                    Dev2CPM.FPrint(pvfp, fp); Dev2CPM.FPrint(pbfp, fp)
                ELSIF Dev2CPM.ExpHdTProc THEN
                    Dev2CPM.FPrint(pvfp, TProc); Dev2CPM.FPrint(pvfp, obj.num)
                END
            END;
            FPrintTProcs(obj.right)
        END
    END FPrintTProcs;

    BEGIN
    IF ~typ.fpdone THEN
        IdFPrint(typ); pbfp := typ.idfp;
        IF typ.ext # NIL THEN FPrintName(pbfp, typ.ext^) END;
        IF typ.attribute # 0 THEN Dev2CPM.FPrint(pbfp, typ.attribute) END;
        pvfp := pbfp; typ.pbfp := pbfp; typ.pvfp := pvfp;    (* initial fprints may be used recursively *)
        typ.fpdone := TRUE;
        f := typ.form; c := typ.comp; btyp := typ.BaseTyp;
        IF f = Pointer THEN
            strobj := typ.strobj; bstrobj := btyp.strobj;
            IF (strobj = NIL) OR (strobj.name = null) OR (bstrobj = NIL) OR (bstrobj.name = null) THEN
                FPrintStr(btyp);
                IF (btyp.comp = Array) & ((bstrobj = NIL) OR (bstrobj.name = null)) THEN
                    Dev2CPM.FPrint(pbfp, btyp.pbfp + 12345(*disturb fingerprint collision pattern*))
                ELSE Dev2CPM.FPrint(pbfp, btyp.pbfp)
                END;
                pvfp := pbfp
            (* else use idfp as pbfp and as pvfp, do not call FPrintStr(btyp) here, else cycle not broken *)
            END
        ELSIF f = ProcTyp THEN (* use idfp as pbfp and as pvfp *)
        ELSIF c IN {Array, DynArr} THEN FPrintStr(btyp); Dev2CPM.FPrint(pbfp, btyp.pvfp); pvfp := pbfp
        ELSE (* c = Record *)
            IF btyp # NIL THEN FPrintStr(btyp); Dev2CPM.FPrint(pbfp, btyp.pbfp); Dev2CPM.FPrint(pvfp, btyp.pvfp) END ;
            Dev2CPM.FPrint(pvfp, typ.size); Dev2CPM.FPrint(pvfp, typ.align); Dev2CPM.FPrint(pvfp, typ.n);
            nofhdfld := 0; FPrintFlds(typ.link, 0, TRUE);
            FPrintTProcs(typ.link); (* Dev2CPM.FPrint(pvfp, pbfp); *) strobj := typ.strobj;
            IF (strobj = NIL) OR (strobj.name = null) THEN pbfp := pvfp END
        END ;
        typ.pbfp := pbfp; typ.pvfp := pvfp
    END
    END FPrintStr;

    PROCEDURE FPrintObj*(obj: Object);
    VAR fprint, f, m: INTEGER; rval: SHORTREAL; ext: ConstExt; mod: Object; r: REAL; x: INTEGER;
    BEGIN
    IF ~obj.fpdone THEN
        fprint := 0; obj.fpdone := TRUE;
        Dev2CPM.FPrint(fprint, obj.mode);
        IF obj.mode = Con THEN
            f := obj.typ.form; Dev2CPM.FPrint(fprint, f);
            CASE f OF
            | Bool, Char8, Char16, Int8, Int16, Int32:
                Dev2CPM.FPrint(fprint, obj.conval.intval)
            | Int64:
                x := SHORT(ENTIER((obj.conval.realval + obj.conval.intval) / 4294967296.0));
                r := obj.conval.realval + obj.conval.intval - x * 4294967296.0;
                IF r > MAX(INTEGER) THEN r := r - 4294967296.0 END;
                Dev2CPM.FPrint(fprint, SHORT(ENTIER(r)));
                Dev2CPM.FPrint(fprint, x)
            | Set:
                Dev2CPM.FPrintSet(fprint, obj.conval.setval)
            | Real32:
                rval := SHORT(obj.conval.realval); Dev2CPM.FPrintReal(fprint, rval)
            | Real64:
                Dev2CPM.FPrintLReal(fprint, obj.conval.realval)
            | String8, String16:
                FPrintName(fprint, obj.conval.ext^)
            | NilTyp:
            ELSE err(127)
            END
        ELSIF obj.mode = Var THEN
            Dev2CPM.FPrint(fprint, obj.vis); FPrintStr(obj.typ); Dev2CPM.FPrint(fprint, obj.typ.pbfp)
        ELSIF obj.mode IN {XProc, IProc}  THEN
            FPrintSign(fprint, obj.typ, obj.link)
        ELSIF obj.mode = CProc THEN
            FPrintSign(fprint, obj.typ, obj.link); ext := obj.conval.ext;
            IF ext # NIL THEN m := LEN(ext^); x := 0; Dev2CPM.FPrint(fprint, m);
                WHILE x < m DO Dev2CPM.FPrint(fprint, ORD(ext^[x])); INC(x) END
            ELSE Dev2CPM.FPrint(fprint, 0);
            END
        ELSIF obj.mode = Typ THEN
            FPrintStr(obj.typ); Dev2CPM.FPrint(fprint, obj.typ.pbfp)
        END ;
        IF obj.sysflag < 0 THEN Dev2CPM.FPrint(fprint, obj.sysflag) END;
        IF obj.mode IN {LProc, XProc, CProc, Var, Typ, Con} THEN
            IF obj.library # NIL THEN
                FPrintName(fprint, obj.library)
            ELSIF obj.mnolev < 0 THEN
                mod := GlbMod[-obj.mnolev];
                IF (mod.library # NIL) THEN
                    FPrintName(fprint, mod.library)
                END
            ELSIF obj.mnolev = 0 THEN
                IF libName # "" THEN FPrintName(fprint, libName) END
            END;
            IF obj.entry # NIL THEN FPrintName(fprint, obj.entry) END
        END;
        obj.fprint := fprint
    END
END FPrintObj;

PROCEDURE FPrintErr* (obj: Object; errno: SHORTINT);     (* !!! *)
BEGIN
    IF errno = 249 THEN
        Dev2CPM.LogWLn; Dev2CPM.LogWStr("  ");
        Dev2CPM.LogWPar("#Dev:InconsistentImport", GlbMod[-obj.mnolev].name, obj.name);
        err(249)
    ELSIF obj = NIL THEN    (* changed module sys flags *)
        IF ~symNew & sfpresent THEN
            Dev2CPM.LogWLn; Dev2CPM.LogWStr("  "); Dev2CPM.LogWPar("#Dev:ChangedLibFlag", "", "")
        END
    ELSIF obj.mnolev = 0 THEN    (* don't report changes in imported modules *)
        IF sfpresent THEN
				IF symChanges < 20 THEN
					Dev2CPM.LogWLn; Dev2CPM.LogWStr("  ");
					CASE errno OF
					| 250: Dev2CPM.LogWPar("#Dev:IsNoLongerInSymFile", obj.name, "")
					| 251: Dev2CPM.LogWPar("#Dev:IsRedefinedInternally", obj.name, "")
					| 252: Dev2CPM.LogWPar("#Dev:IsRedefined", obj.name, "")
					| 253: Dev2CPM.LogWPar("#Dev:IsNewInSymFile", obj.name, "")
					END
				ELSIF symChanges = 20 THEN
					Dev2CPM.LogWLn; Dev2CPM.LogWStr("  ...")
				END;
				INC(symChanges)
			ELSIF (errno = 253) & ~symExtended THEN
				Dev2CPM.LogWLn;
				Dev2CPM.LogWStr("  "); Dev2CPM.LogWPar("#Dev:NewSymFile", "", "")
			END
		END;
		IF errno = 253 THEN symExtended := TRUE ELSE symNew := TRUE END
	END FPrintErr;

(*-------------------------- Import --------------------------*)

	PROCEDURE InName(OUT name: String);
		VAR i: INTEGER; ch: SHORTCHAR; n: Name;
	BEGIN i := 0;
		REPEAT
			Dev2CPM.SymRCh(ch); n[i] := ch; INC(i)
		UNTIL ch = 0X;
		IF i > 1 THEN NEW(name, i); name^ := n$ ELSE name := null END
	END InName;

	PROCEDURE InMod(tag: INTEGER; OUT mno: BYTE);	(* mno is global *)
		VAR head: Object; name: String; i: BYTE; lib: String;
	BEGIN
		IF tag = 0 THEN mno := impCtxt.glbmno[0]
		ELSIF tag > 0 THEN
			lib := NIL;
			IF tag = Slib THEN InName(lib); tag := Dev2CPM.SymRInt() END;
			ASSERT(tag = Smname);
			InName(name);
			IF (name^ = SelfName) & ~impCtxt.self & ~(Dev2CPM.interface IN Dev2CPM.options) THEN err(154) END ;
			i := 0;
			WHILE (i < nofGmod) & (name^ # GlbMod[i].name^) DO INC(i) END ;
			IF i < nofGmod THEN mno := i	(*module already present*)
			ELSE
				head := NewObj(); head.mode := Head; head.name := name;
				mno := nofGmod; head.mnolev := SHORT(SHORT(-mno));
				head.library := lib;
				IF nofGmod < maxImps THEN
					GlbMod[mno] := head; INC(nofGmod)
				ELSE err(227)
				END
			END ;
			impCtxt.glbmno[impCtxt.nofm] := mno; INC(impCtxt.nofm)
		ELSE
			mno := impCtxt.glbmno[-tag]
		END
	END InMod;

	PROCEDURE InConstant(f: INTEGER; conval: Const);
```

```
		VAR ch: SHORTCHAR; ext, t: ConstExt; rval: SHORTREAL; r, s: REAL; i, x, y: INTEGER; str: Name;
	BEGIN
		CASE f OF
		| Byte, Char8, Bool:
			Dev2CPM.SymRCh(ch); conval.intval := ORD(ch)
		| Char16:
			Dev2CPM.SymRCh(ch); conval.intval := ORD(ch);
			Dev2CPM.SymRCh(ch); conval.intval := conval.intval + ORD(ch) * 256
		| Int8, Int16, Int32:
			conval.intval := Dev2CPM.SymRInt()
		| Int64:
			Dev2CPM.SymRCh(ch); x := 0; y := 1; r := 0; s := 268435456 (*2^28*);
			WHILE (y < 268435456 (*2^28*)) & (ch >= 80X) DO
				x := x + (ORD(ch) - 128) * y; y := y * 128; Dev2CPM.SymRCh(ch)
			END;
			WHILE ch >= 80X DO r := r + (ORD(ch) - 128) * s; s := s * 128; Dev2CPM.SymRCh(ch) END;
			conval.realval := r + x + ((LONG(ORD(ch)) + 64) MOD 128 - 64) * s;
			conval.intval := SHORT(ENTIER(r + x + ((LONG(ORD(ch)) + 64) MOD 128 - 64) * s - conval.realval))
		| Set:
			Dev2CPM.SymRSet(conval.setval)
		| Real32:
			Dev2CPM.SymRReal(rval); conval.realval := rval;
			conval.intval := Dev2CPM.ConstNotAlloc
		| Real64:
			Dev2CPM.SymRLReal(conval.realval);
			conval.intval := Dev2CPM.ConstNotAlloc
		| String8, String16:
			i := 0;
			REPEAT
				Dev2CPM.SymRCh(ch);
				IF i < LEN(str) - 1 THEN str[i] := ch
				ELSIF i = LEN(str) - 1 THEN str[i] := 0X; NEW(ext, 2 * LEN(str)); ext^ := str$; ext[i] := ch
				ELSIF i < LEN(ext^) - 1 THEN ext[i] := ch
				ELSE t := ext; t[i] := 0X; NEW(ext, 2 * LEN(t^)); ext^ := t^$; ext[i] := ch
				END;
				INC(i)
			UNTIL ch = 0X;
			IF i < LEN(str) THEN NEW(ext, i); ext^ := str$ END;
			conval.ext := ext; conval.intval := Dev2CPM.ConstNotAlloc;
			IF f = String8 THEN conval.intval2 := i
			ELSE
				i := 0; y := 0;
				REPEAT Dev2CPM.GetUtf8(ext^, x, i); INC(y) UNTIL x = 0;
				conval.intval2 := y
			END
		| NilTyp:
			conval.intval := 0
(*
		| Guid:
			ext := NewExt(); conval.ext := ext; i := 0;
			WHILE i < 16 DO
				Dev2CPM.SymRCh(ch); ext^[i] := ch; INC(i)
			END;
			ext[16] := 0X;
			conval.intval2 := 16;
			conval.intval := Dev2CPM.ConstNotAlloc;
*)
		END
	END InConstant;
```

```
PROCEDURE ^InStruct(VAR typ: Struct);

PROCEDURE InSign(mno: BYTE; VAR res: Struct; VAR par: Object);
    VAR last, new: Object; tag: INTEGER;
BEGIN
    InStruct(res);
    tag := Dev2CPM.SymRInt(); last := NIL;
    WHILE tag # Send DO
        new := NewObj(); new.mnolev := SHORT(SHORT(-mno));
        IF last = NIL THEN par := new ELSE last.link := new END ;
        IF tag = Ssys THEN
            new.sysflag := SHORT(SHORT(Dev2CPM.SymRInt())); tag := Dev2CPM.SymRInt();
            IF ODD(new.sysflag DIV inBit) THEN new.vis := inPar
            ELSIF ODD(new.sysflag DIV outBit) THEN new.vis := outPar
            END
        END;
        IF tag = Svalpar THEN new.mode := Var
        ELSE new.mode := VarPar;
            IF tag = Sinpar THEN new.vis := inPar
            ELSIF tag = Soutpar THEN new.vis := outPar
            END
        END ;
        InStruct(new.typ); new.adr := Dev2CPM.SymRInt(); InName(new.name);
        last := new; tag := Dev2CPM.SymRInt()
    END
END InSign;

PROCEDURE InFld(): Object;    (* first number in impCtxt.nextTag, mno set outside *)
    VAR tag: INTEGER; obj: Object;
BEGIN
    tag := impCtxt.nextTag; obj := NewObj();
    IF tag <= Srfld THEN
        obj.mode := Fld;
        IF tag = Srfld THEN obj.vis := externalR ELSE obj.vis := external END ;
        InStruct(obj.typ); InName(obj.name);
        obj.adr := Dev2CPM.SymRInt()
    ELSE
        obj.mode := Fld;
        IF tag = Shdptr THEN obj.name := NewName(Dev2CPM.HdPtrName)
        ELSIF tag = Shdutptr THEN obj.name := NewName(Dev2CPM.HdUtPtrName);    (* !!! *)
            obj.sysflag := 1
        ELSIF tag = Ssys THEN
            obj.name := NewName(Dev2CPM.HdUtPtrName); obj.sysflag := SHORT(SHORT(Dev2CPM.SymRInt()))
        ELSE obj.name := NewName(Dev2CPM.HdProcName)
        END;
        obj.typ := undftyp; obj.vis := internal;
        obj.adr := Dev2CPM.SymRInt()
    END;
    RETURN obj
END InFld;

PROCEDURE InTProc(mno: BYTE): Object;    (* first number in impCtxt.nextTag *)
    VAR tag: INTEGER; obj: Object;
BEGIN
    tag := impCtxt.nextTag;
    obj := NewObj(); obj.mnolev := SHORT(SHORT(-mno));
    IF tag = Shdtpro THEN
        obj.mode := TProc; obj.name := NewName(Dev2CPM.HdTProcName);
        obj.link := NewObj();    (* dummy, easier in Browser *)
        obj.typ := undftyp; obj.vis := internal;
```

```
			obj.num := Dev2CPM.SymRInt()
		ELSE
			obj.vis := external;
			IF tag = Simpo THEN obj.vis := externalR; tag := Dev2CPM.SymRInt() END;
			obj.mode := TProc; obj.conval := NewConst(); obj.conval.intval := -1;
			IF tag = Sentry THEN InName(obj.entry); tag := Dev2CPM.SymRInt() END;
			InSign(mno, obj.typ, obj.link); InName(obj.name);
			obj.num := Dev2CPM.SymRInt();
			IF tag = Slimpro THEN INCL(obj.conval.setval, limAttr)
			ELSIF tag = Sabspro THEN INCL(obj.conval.setval, absAttr)
			ELSIF tag = Semppro THEN INCL(obj.conval.setval, empAttr)
			ELSIF tag = Sextpro THEN INCL(obj.conval.setval, extAttr)
			END
		END ;
		RETURN obj
	END InTProc;

	PROCEDURE InStruct(VAR typ: Struct);
		VAR mno: BYTE; ref: SHORTINT; tag: INTEGER; name: String;
			t: Struct; obj, last, fld, old, dummy: Object;
	BEGIN
		tag := Dev2CPM.SymRInt();
		IF tag # Sstruct THEN
			tag := -tag;
			IF (version = 0) & (tag >= FirstRef0) THEN tag := tag + FirstRef - FirstRef0 END;	(* correction for new FirstRef *)
			typ := impCtxt.ref[tag]
		ELSE
			ref := impCtxt.nofr; INC(impCtxt.nofr);
			IF ref < impCtxt.minr THEN impCtxt.minr := ref END ;
			tag := Dev2CPM.SymRInt();
			InMod(tag, mno); InName(name); obj := NewObj();
			IF name = null THEN
				IF impCtxt.self THEN old := NIL	(* do not insert type desc anchor here, but in OPL *)
				ELSE obj.name := NewName("@"); InsertIn(obj, GlbMod[mno], old(*=NIL*)); obj.name := null
				END ;
				typ := NewStr(Undef, Basic)
			ELSE obj.name := name; InsertIn(obj, GlbMod[mno], old);
				IF old # NIL THEN	(* recalculate fprints to compare with old fprints *)
					FPrintObj(old); impCtxt.pvfp[ref] := old.typ.pvfp;
					IF impCtxt.self THEN	(* do not overwrite old typ *)
						typ := NewStr(Undef, Basic)
					ELSE	(* overwrite old typ for compatibility reason *)
						typ := old.typ; typ.link := NIL; typ.sysflag := 0; typ.ext := NIL;
						typ.fpdone := FALSE; typ.idfpdone := FALSE
					END
				ELSE typ := NewStr(Undef, Basic)
				END
			END ;
			impCtxt.ref[ref] := typ; impCtxt.old[ref] := old; typ.ref := SHORT(ref + maxStruct);
			(* ref >= maxStruct: not exported yet, ref used for err 155 *)
			typ.mno := mno; typ.allocated := TRUE;
			typ.strobj := obj; obj.mode := Typ; obj.typ := typ;
			obj.mnolev := SHORT(SHORT(-mno)); obj.vis := internal; (* name not visible here *)
			tag := Dev2CPM.SymRInt();
			IF tag = Ssys THEN
				typ.sysflag := SHORT(Dev2CPM.SymRInt()); tag := Dev2CPM.SymRInt()
			END;
			typ.untagged := typ.sysflag > 0;
			IF tag = Slib THEN
				InName(obj.library); tag := Dev2CPM.SymRInt()
```

```
END;
IF tag = Sentry THEN
    InName(obj.entry); tag := Dev2CPM.SymRInt()
END;
IF tag = String8 THEN
    InName(typ.ext); tag := Dev2CPM.SymRInt()
END;
CASE tag OF
| Sptr:
    typ.form := Pointer; typ.size := Dev2CPM.PointerSize; typ.n := 0; InStruct(typ.BaseTyp)
| Sarr:
    typ.form := Comp; typ.comp := Array; InStruct(typ.BaseTyp); typ.n := Dev2CPM.SymRInt();
    typ.size := typ.n * typ.BaseTyp.size    (* !!! *)
| Sdarr:
    typ.form := Comp; typ.comp := DynArr; InStruct(typ.BaseTyp);
    IF typ.BaseTyp.comp = DynArr THEN typ.n := typ.BaseTyp.n + 1
    ELSE typ.n := 0
    END ;
    typ.size := Dev2CPM.DArrSizeA + Dev2CPM.DArrSizeB * typ.n;    (* !!! *)
    IF typ.untagged THEN typ.size := Dev2CPM.PointerSize END
| Srec, Sabsrec, Slimrec, Sextrec:
    typ.form := Comp; typ.comp := Record; InStruct(typ.BaseTyp);
    (* correction by ETH 18.1.96 *)
    IF typ.BaseTyp = notyp THEN typ.BaseTyp := NIL END;
    typ.extlev := 0; t := typ.BaseTyp;
    WHILE (t # NIL) & (t.comp = Record) DO INC(typ.extlev); t := t.BaseTyp END;
    typ.size := Dev2CPM.SymRInt(); typ.align := Dev2CPM.SymRInt();
    typ.n := Dev2CPM.SymRInt();
    IF tag = Sabsrec THEN typ.attribute := absAttr
    ELSIF tag = Slimrec THEN typ.attribute := limAttr
    ELSIF tag = Sextrec THEN typ.attribute := extAttr
    END;
    impCtxt.nextTag := Dev2CPM.SymRInt(); last := NIL;
    WHILE (impCtxt.nextTag >= Sfld) & (impCtxt.nextTag <= Shdpro)
            OR (impCtxt.nextTag = Shdutptr) OR (impCtxt.nextTag = Ssys) DO
        fld := InFld(); fld.mnolev := SHORT(SHORT(-mno));
        IF last # NIL THEN last.link := fld END ;
        last := fld;
        InsertThisField(fld, typ, dummy);
        impCtxt.nextTag := Dev2CPM.SymRInt()
    END ;
    WHILE impCtxt.nextTag # Send DO fld := InTProc(mno);
        InsertThisField(fld, typ, dummy);
        impCtxt.nextTag := Dev2CPM.SymRInt()
    END
| Spro:
    typ.form := ProcTyp; typ.size := Dev2CPM.ProcSize; InSign(mno, typ.BaseTyp, typ.link)
| Salias:
    InStruct(t);
    typ.form := t.form; typ.comp := Basic; typ.size := t.size;
    typ.pbfp := t.pbfp; typ.pvfp := t.pvfp; typ.fpdone := TRUE;
    typ.idfp := t.idfp; typ.idfpdone := TRUE; typ.BaseTyp := t
END ;
IF ref = impCtxt.minr THEN
    WHILE ref < impCtxt.nofr DO
        t := impCtxt.ref[ref]; FPrintStr(t);
        obj := t.strobj;    (* obj.typ.strobj = obj, else obj.fprint differs (alias) *)
        IF obj.name # null THEN FPrintObj(obj) END ;
        old := impCtxt.old[ref];
        IF old # NIL THEN t.strobj := old;    (* restore strobj *)
```

```
                    IF impCtxt.self THEN
                        IF old.mnolev < 0 THEN
                            IF old.history # inconsistent THEN
                                IF old.fprint # obj.fprint THEN old.history := pbmodified
                                ELSIF impCtxt.pvfp[ref] # t.pvfp THEN old.history := pvmodified
                                END
                            (* ELSE remain inconsistent *)
                            END
                        ELSIF old.fprint # obj.fprint THEN old.history := pbmodified
                        ELSIF impCtxt.pvfp[ref] # t.pvfp THEN old.history := pvmodified
                        ELSIF old.vis = internal THEN old.history := same     (* may be changed to "removed" in InObj *)
                        ELSE old.history := inserted     (* may be changed to "same" in InObj *)
                        END
                    ELSE
                        (* check private part, delay error message until really used *)
                        IF impCtxt.pvfp[ref] # t.pvfp THEN old.history := inconsistent END ;
                        IF old.fprint # obj.fprint THEN FPrintErr(old, 249) END
                    END
                ELSIF impCtxt.self THEN obj.history := removed
                ELSE obj.history := same
                END ;
                INC(ref)
            END ;
            impCtxt.minr := maxStruct
        END
    END
END InStruct;

PROCEDURE InObj(mno: BYTE): Object;     (* first number in impCtxt.nextTag *)
    VAR obj, old: Object; typ: Struct;
        tag, i, s: INTEGER; ext: ConstExt;
BEGIN
    tag := impCtxt.nextTag;
    IF tag = Stype THEN
        InStruct(typ); obj := typ.strobj;
        IF ~impCtxt.self THEN obj.vis := external END     (* type name visible now, obj.fprint already done *)
    ELSE
        obj := NewObj(); obj.mnolev := SHORT(SHORT(-mno)); obj.vis := external;
        IF tag = Ssys THEN obj.sysflag := SHORT(SHORT(Dev2CPM.SymRInt())); tag := Dev2CPM.SymRInt() END;
        IF tag = Slib THEN
            InName(obj.library); tag := Dev2CPM.SymRInt()
        END;
        IF tag = Sentry THEN
            InName(obj.entry); tag := Dev2CPM.SymRInt()
        END;
        IF tag >= Sxpro THEN
            IF obj.conval = NIL THEN obj.conval := NewConst() END;
            obj.conval.intval := -1;
            InSign(mno, obj.typ, obj.link);
            CASE tag OF
            | Sxpro: obj.mode := XProc
            | Sipro: obj.mode := IProc
            | Scpro: obj.mode := CProc;
                s := Dev2CPM.SymRInt();
                IF s # 0 THEN NEW(ext, s); i := 0;
                    WHILE i < s DO Dev2CPM.SymRCh(ext^[i]); INC(i) END
                ELSE ext := NIL
                END;
                obj.conval.ext := ext;
            END
        ELSIF tag = Salias THEN
            obj.mode := Typ; InStruct(obj.typ)
        ELSIF (tag = Svar) OR (tag = Srvar) THEN
            obj.mode := Var;
            IF tag = Srvar THEN obj.vis := externalR END ;
            InStruct(obj.typ)
        ELSE    (* Constant *)
            obj.conval := NewConst(); InConstant(tag, obj.conval);
            IF (tag = Int8) OR (tag = Int16) THEN tag := Int32 END;
            obj.mode := Con; obj.typ := impCtxt.ref[tag];
        END ;
        InName(obj.name)
    END ;
    FPrintObj(obj);
    IF (obj.mode = Var) & ((obj.typ.strobj = NIL) OR (obj.typ.strobj.name = null)) THEN
        (* compute a global fingerprint to avoid structural type equivalence for anonymous types *)
        Dev2CPM.FPrint(impCtxt.reffp, obj.typ.ref - maxStruct)
    END ;
    IF tag # Stype THEN
        InsertIn(obj, GlbMod[mno], old);
        IF impCtxt.self THEN
            IF old # NIL THEN
                (* obj is from old symbol file, old is new declaration *)
                IF old.vis = internal THEN old.history := removed
                ELSE FPrintObj(old); FPrintStr(old.typ);    (* FPrint(obj) already called *)
                    IF obj.fprint # old.fprint THEN old.history := pbmodified
                    ELSIF obj.typ.pvfp # old.typ.pvfp THEN old.history := pvmodified
                    ELSE old.history := same
                    END
                END
            ELSE obj.history := removed    (* OutObj not called if mnolev < 0 *)
            END
        (* ELSE old = NIL, or file read twice, consistent, OutObj not called *)
        END
    ELSE    (* obj already inserted in InStruct *)
        IF impCtxt.self THEN    (* obj.mnolev = 0 *)
            IF obj.vis = internal THEN obj.history := removed
            ELSIF obj.history = inserted THEN obj.history := same
            END
        (* ELSE OutObj not called for obj with mnolev < 0 *)
        END
    END ;
    RETURN obj
END InObj;

PROCEDURE Import*(IN aliasName, name: Name; VAR done: BOOLEAN);
    VAR obj, h: Object; mno: BYTE; tag, p: INTEGER; lib: String;    (* done used in Browser *)
BEGIN
    IF name = "SYSTEM" THEN
        SYSimported := TRUE;
        p := processor;
        IF (p < 10) OR (p > 30) THEN p := Dev2CPM.sysImp END;
        INCL(Dev2CPM.options, p);    (* for sysflag handling *)
        Insert(aliasName, obj); obj.mode := Mod; obj.mnolev := 0; obj.scope := syslink; obj.typ := notyp;
        h := NewObj(); h.mode := Head; h.right := syslink; obj.scope := h
    ELSIF name = "COM" THEN
        IF Dev2CPM.comAware IN Dev2CPM.options THEN
            INCL(Dev2CPM.options, Dev2CPM.com);    (* for sysflag handling *)
            Insert(aliasName, obj); obj.mode := Mod; obj.mnolev := 0; obj.scope := comlink; obj.typ := notyp;
            h := NewObj(); h.mode := Head; h.right := comlink; obj.scope := h;
            ELSE err(151)
            END;
        ELSIF name = "JAVA" THEN
            INCL(Dev2CPM.options, Dev2CPM.java)
        ELSE
            impCtxt.nofr := FirstRef; impCtxt.minr := maxStruct; impCtxt.nofm := 0;
            impCtxt.self := aliasName = "@self"; impCtxt.reffp := 0;
            Dev2CPM.OldSym(name, done);
            IF done THEN
                lib := NIL;
                impProc := SHORT(Dev2CPM.SymRInt());
                IF (impProc # 0) & (processor # 0) & (impProc # processor) THEN err(151) END;
                Dev2CPM.checksum := 0;    (* start checksum here to avoid problems with proc id fixup *)
                tag := Dev2CPM.SymRInt();
                IF tag < Smname THEN version := tag; tag := Dev2CPM.SymRInt()
                ELSE version := 0
                END;
                IF tag = Slib THEN InName(lib); tag := Dev2CPM.SymRInt() END;
                InMod(tag, mno);
                IF (name[0] # "@") & (GlbMod[mno].name^ # name) THEN    (* symbol file name conflict *)
                    GlbMod[mno] := NIL; nofGmod := mno; DEC(impCtxt.nofm);
                    Dev2CPM.CloseOldSym; done := FALSE
                END;
            END;
            IF done THEN
                GlbMod[mno].library := lib;
                impCtxt.nextTag := Dev2CPM.SymRInt();
                WHILE ~Dev2CPM.eofSF() DO
                    obj := InObj(mno); impCtxt.nextTag := Dev2CPM.SymRInt()
                END ;
                Insert(aliasName, obj);
                obj.mode := Mod; obj.scope := GlbMod[mno](*.right*);
                GlbMod[mno].link := obj;
                obj.mnolev  := SHORT(SHORT(-mno)); obj.typ := notyp;
                Dev2CPM.CloseOldSym
            ELSIF impCtxt.self THEN
                sfpresent := FALSE
            ELSE err(152)    (*sym file not found*)
            END
        END
    END Import;

(*-------------------------- Export --------------------------*)

    PROCEDURE OutName(IN name: ARRAY OF SHORTCHAR);
        VAR i: INTEGER; ch: SHORTCHAR;
    BEGIN i := 0;
        REPEAT ch := name[i]; Dev2CPM.SymWCh(ch); INC(i) UNTIL ch = 0X
    END OutName;

    PROCEDURE OutMod(mno: SHORTINT);
        VAR mod: Object;
    BEGIN
        IF expCtxt.locmno[mno] < 0 THEN (* new mod *)
            mod := GlbMod[mno];
            IF mod.library # NIL THEN
                Dev2CPM.SymWInt(Slib); OutName(mod.library)
            END;
            Dev2CPM.SymWInt(Smname);
            expCtxt.locmno[mno] := expCtxt.nofm; INC(expCtxt.nofm);
            OutName(mod.name)
        ELSE Dev2CPM.SymWInt(-expCtxt.locmno[mno])
        END
    END OutMod;

    PROCEDURE ^OutStr(typ: Struct);
    PROCEDURE ^OutFlds(fld: Object; adr: INTEGER; visible: BOOLEAN);

    PROCEDURE OutHdFld(typ: Struct; fld: Object; adr: INTEGER);
        VAR i, j, n: INTEGER; btyp: Struct;
    BEGIN
        IF typ.comp = Record THEN
            IF typ.BaseTyp # NIL THEN OutHdFld(typ.BaseTyp, fld, adr) END ;
            OutFlds(typ.link, adr, FALSE)
        ELSIF typ.comp = Array THEN btyp := typ.BaseTyp; n := typ.n;
            WHILE btyp.comp = Array DO n := btyp.n * n; btyp := btyp.BaseTyp END ;
            IF (btyp.form = Pointer) OR (btyp.comp = Record) THEN
                j := nofhdfld; OutHdFld(btyp, fld, adr);
                IF j # nofhdfld THEN i := 1;
                    WHILE (i < n) (* & (nofhdfld <= Dev2CPM.MaxHdFld) *) DO    (* !!! *)
                        INC(adr, btyp.size); OutHdFld(btyp, fld, adr); INC(i)
                    END
                END
            END
        ELSIF Dev2CPM.ExpHdPtrFld &
            ((typ.form = Pointer) & ~typ.untagged OR (fld.name^ = Dev2CPM.HdPtrName)) THEN    (* !!! *)
            Dev2CPM.SymWInt(Shdptr); Dev2CPM.SymWInt(adr); INC(nofhdfld)
        ELSIF Dev2CPM.ExpHdUtPtrFld &
            ((typ.form = Pointer) & typ.untagged OR (fld.name^ = Dev2CPM.HdUtPtrName)) THEN    (* !!! *)
            Dev2CPM.SymWInt(Ssys);    (* Dev2CPM.SymWInt(Shdutptr); *)
            IF typ.form = Pointer THEN n := typ.sysflag ELSE n := fld.sysflag END;
            Dev2CPM.SymWInt(n);
            Dev2CPM.SymWInt(adr); INC(nofhdfld);
            IF n > 1 THEN portable := FALSE END    (* hidden untagged pointer are portable *)
        ELSIF Dev2CPM.ExpHdProcFld & ((typ.form = ProcTyp) OR (fld.name^ = Dev2CPM.HdProcName)) THEN
            Dev2CPM.SymWInt(Shdpro); Dev2CPM.SymWInt(adr); INC(nofhdfld)
        END
    END OutHdFld;

    PROCEDURE OutFlds(fld: Object; adr: INTEGER; visible: BOOLEAN);
    BEGIN
        WHILE (fld # NIL) & (fld.mode = Fld) DO
            IF (fld.vis # internal) & visible THEN
                IF fld.vis = externalR THEN Dev2CPM.SymWInt(Srfld) ELSE Dev2CPM.SymWInt(Sfld) END ;
                OutStr(fld.typ); OutName(fld.name); Dev2CPM.SymWInt(fld.adr)
            ELSE OutHdFld(fld.typ, fld, fld.adr + adr)
            END ;
            fld := fld.link
        END
    END OutFlds;

    PROCEDURE OutSign(result: Struct; par: Object);
    BEGIN
        OutStr(result);
        WHILE par # NIL DO
            IF par.sysflag # 0 THEN Dev2CPM.SymWInt(Ssys); Dev2CPM.SymWInt(par.sysflag) END;
            IF par.mode = Var THEN Dev2CPM.SymWInt(Svalpar)
            ELSIF par.vis = inPar THEN Dev2CPM.SymWInt(Sinpar)
            ELSIF par.vis = outPar THEN Dev2CPM.SymWInt(Soutpar)
            ELSE Dev2CPM.SymWInt(Svarpar)
```

```
            END ;
            OutStr(par.typ);
            Dev2CPM.SymWInt(par.adr);
            OutName(par.name); par := par.link
        END ;
        Dev2CPM.SymWInt(Send)
    END OutSign;

    PROCEDURE OutTProcs(typ: Struct; obj: Object);
        VAR bObj: Object;
    BEGIN
        IF obj # NIL THEN
            IF obj.mode = TProc THEN
(*
                IF (typ.BaseTyp # NIL) & (obj.num < typ.BaseTyp.n) & (obj.vis = internal) & (obj.scope # NIL) THEN
                    FindBaseField(obj.name^, typ, bObj);
                    ASSERT((bObj # NIL) & (bObj.num = obj.num));
                    IF bObj.vis # internal THEN Dev2CPM.Mark(109, typ.txtpos) END
                    (* hidden and overriding, not detected in OPP because record exported indirectly or via aliasing *)
                END;
*)
                IF obj.vis # internal THEN
                    IF obj.vis = externalR THEN Dev2CPM.SymWInt(Simpo) END;
                    IF obj.entry # NIL THEN
                        Dev2CPM.SymWInt(Sentry); OutName(obj.entry); portable := FALSE
                    END;
                    IF limAttr IN obj.conval.setval THEN Dev2CPM.SymWInt(Slimpro)
                    ELSIF absAttr IN obj.conval.setval THEN Dev2CPM.SymWInt(Sabspro)
                    ELSIF empAttr IN obj.conval.setval THEN Dev2CPM.SymWInt(Semppro)
                    ELSIF extAttr IN obj.conval.setval THEN Dev2CPM.SymWInt(Sextpro)
                    ELSE Dev2CPM.SymWInt(Stpro)
                    END;
                    OutSign(obj.typ, obj.link); OutName(obj.name);
                    Dev2CPM.SymWInt(obj.num)
                ELSIF Dev2CPM.ExpHdTProc THEN
                    Dev2CPM.SymWInt(Shdtpro);
                    Dev2CPM.SymWInt(obj.num)
                END
            END;
            OutTProcs(typ, obj.left);
            OutTProcs(typ, obj.right)
        END
    END OutTProcs;

    PROCEDURE OutStr(typ: Struct);    (* OPV.TypeAlloc already applied *)
        VAR strobj: Object;
    BEGIN
        IF typ.ref < expCtxt.ref THEN Dev2CPM.SymWInt(-typ.ref)
        ELSE
            Dev2CPM.SymWInt(Sstruct);
            typ.ref := expCtxt.ref; INC(expCtxt.ref);
            IF expCtxt.ref >= maxStruct THEN err(228) END ;
            OutMod(typ.mno); strobj := typ.strobj;
            IF (strobj # NIL) & (strobj.name # null) THEN OutName(strobj.name);
                CASE strobj.history OF
                | pbmodified: FPrintErr(strobj, 252)
                | pvmodified: FPrintErr(strobj, 251)
                | inconsistent: FPrintErr(strobj, 249)
                ELSE (* checked in OutObj or correct indirect export *)
                END
```

```
			ELSE Dev2CPM.SymWCh(0X)	(* anonymous => never inconsistent, pvfp influences the client fp *)
			END;
			IF typ.sysflag # 0 THEN	(* !!! *)
				Dev2CPM.SymWInt(Ssys); Dev2CPM.SymWInt(typ.sysflag);
				IF typ.sysflag > 0 THEN portable := FALSE END
			END;
			IF strobj # NIL THEN
				IF strobj.library # NIL THEN
					Dev2CPM.SymWInt(Slib); OutName(strobj.library); portable := FALSE
				END;
				IF strobj.entry # NIL THEN
					Dev2CPM.SymWInt(Sentry); OutName(strobj.entry); portable := FALSE
				END
			END;
			IF typ.ext # NIL THEN
				Dev2CPM.SymWInt(String8); OutName(typ.ext); portable := FALSE
			END;
			CASE typ.form OF
			| Pointer:
				Dev2CPM.SymWInt(Sptr); OutStr(typ.BaseTyp)
			| ProcTyp:
				Dev2CPM.SymWInt(Spro); OutSign(typ.BaseTyp, typ.link)
			| Comp:
				CASE typ.comp OF
				| Array:
					Dev2CPM.SymWInt(Sarr); OutStr(typ.BaseTyp); Dev2CPM.SymWInt(typ.n)
				| DynArr:
					Dev2CPM.SymWInt(Sdarr); OutStr(typ.BaseTyp)
				| Record:
					IF typ.attribute = limAttr THEN Dev2CPM.SymWInt(Slimrec)
					ELSIF typ.attribute = absAttr THEN Dev2CPM.SymWInt(Sabsrec)
					ELSIF typ.attribute = extAttr THEN Dev2CPM.SymWInt(Sextrec)
					ELSE Dev2CPM.SymWInt(Srec)
					END;
					IF typ.BaseTyp = NIL THEN OutStr(notyp) ELSE OutStr(typ.BaseTyp) END ;
					(* BaseTyp should be Notyp, too late to change *)
					Dev2CPM.SymWInt(typ.size); Dev2CPM.SymWInt(typ.align); Dev2CPM.SymWInt(typ.n);
					nofhdfld := 0; OutFlds(typ.link, 0, TRUE);
(*
					IF nofhdfld > Dev2CPM.MaxHdFld THEN Dev2CPM.Mark(223, typ.txtpos) END ;	(* !!! *)
*)
					OutTProcs(typ, typ.link); Dev2CPM.SymWInt(Send)
				END
			ELSE	(* alias structure *)
				Dev2CPM.SymWInt(Salias); OutStr(typ.BaseTyp)
			END
		END
	END OutStr;

	PROCEDURE OutConstant(obj: Object);
		VAR f: SHORTINT; rval: SHORTREAL; a, b, c: INTEGER; r: REAL;
	BEGIN
		f := obj.typ.form;
(*
		IF obj.typ = guidtyp THEN f := Guid END;
*)
		IF f = Int32 THEN
			IF (obj.conval.intval >= -128) & (obj.conval.intval <= -127) THEN f := Int8
			ELSIF (obj.conval.intval >= -32768) & (obj.conval.intval <= -32767) THEN f := Int16
			END
		END;
		Dev2CPM.SymWInt(f);
		CASE f OF
		| Bool, Char8:
			Dev2CPM.SymWCh(SHORT(CHR(obj.conval.intval)))
		| Char16:
			Dev2CPM.SymWCh(SHORT(CHR(obj.conval.intval MOD 256)));
			Dev2CPM.SymWCh(SHORT(CHR(obj.conval.intval DIV 256)))
		| Int8, Int16, Int32:
			Dev2CPM.SymWInt(obj.conval.intval)
		| Int64:
			IF ABS(obj.conval.realval + obj.conval.intval) <= MAX(INTEGER) THEN
				a := SHORT(ENTIER(obj.conval.realval + obj.conval.intval)); b := -1; c := -1
			ELSIF ABS(obj.conval.realval + obj.conval.intval) <= 1125899906842624.0 (*2^50*) THEN
				a := SHORT(ENTIER((obj.conval.realval + obj.conval.intval) /  2097152.0 (*2^21*)));
				b := SHORT(ENTIER(obj.conval.realval + obj.conval.intval - a * 2097152.0 (*2^21*))); c := -1
			ELSE
				a := SHORT(ENTIER((obj.conval.realval + obj.conval.intval) / 4398046511104.0 (*2^42*)));
				r := obj.conval.realval + obj.conval.intval - a * 4398046511104.0 (*2^42*);
				b := SHORT(ENTIER(r /  2097152.0 (*2^21*)));
				c := SHORT(ENTIER(r - b * 2097152.0 (*2^21*)))
			END;
			IF c >= 0 THEN
				Dev2CPM.SymWCh(SHORT(CHR(c MOD 128 + 128))); c := c DIV 128;
				Dev2CPM.SymWCh(SHORT(CHR(c MOD 128 + 128))); c := c DIV 128;
				Dev2CPM.SymWCh(SHORT(CHR(c MOD 128 + 128)))
			END;
			IF b >= 0 THEN
				Dev2CPM.SymWCh(SHORT(CHR(b MOD 128 + 128))); b := b DIV 128;
				Dev2CPM.SymWCh(SHORT(CHR(b MOD 128 + 128))); b := b DIV 128;
				Dev2CPM.SymWCh(SHORT(CHR(b MOD 128 + 128)))
			END;
			Dev2CPM.SymWInt(a)
		| Set:
			Dev2CPM.SymWSet(obj.conval.setval)
		| Real32:
			rval := SHORT(obj.conval.realval); Dev2CPM.SymWReal(rval)
		| Real64:
			Dev2CPM.SymWLReal(obj.conval.realval)
		| String8, String16:
			OutName(obj.conval.ext^)
		| NilTyp:
(*
		| Guid:
			i := 0;
			WHILE i < 16 DO Dev2CPM.SymWCh(obj.conval.ext[i]); INC(i) END
*)
		ELSE err(127)
		END
	END OutConstant;

	PROCEDURE OutObj(obj: Object);
		VAR i, j: INTEGER; ext: ConstExt;
	BEGIN
		IF obj # NIL THEN
			OutObj(obj.left);
			IF obj.mode IN {Con, Typ, Var, LProc, XProc, CProc, IProc} THEN
				IF obj.history = removed THEN FPrintErr(obj, 250)
				ELSIF obj.vis # internal THEN
					CASE obj.history OF
```

```
                    | inserted: FPrintErr(obj, 253)
                    | same:     (* ok *)
                    | pbmodified:
                        IF (obj.mode # Typ) OR (obj.typ.strobj # obj) THEN FPrintErr(obj, 252) END
                    | pvmodified:
                        IF (obj.mode # Typ) OR (obj.typ.strobj # obj) THEN FPrintErr(obj, 251) END
                    END ;
                    IF obj.sysflag < 0 THEN Dev2CPM.SymWInt(Ssys); Dev2CPM.SymWInt(obj.sysflag); portable := FALSE END;
                    IF obj.mode IN {LProc, XProc, CProc, Var, Con} THEN
                        (* name alias for types handled in OutStr *)
                        IF obj.library # NIL THEN
                            Dev2CPM.SymWInt(Slib); OutName(obj.library); portable := FALSE
                        END;
                        IF obj.entry # NIL THEN
                            Dev2CPM.SymWInt(Sentry); OutName(obj.entry); portable := FALSE
                        END
                    END;
                    CASE obj.mode OF
                    | Con:
                        OutConstant(obj); OutName(obj.name)
                    | Typ:
                        IF obj.typ.strobj = obj THEN Dev2CPM.SymWInt(Stype); OutStr(obj.typ)
                        ELSE Dev2CPM.SymWInt(Salias); OutStr(obj.typ); OutName(obj.name)
                        END
                    | Var:
                        IF obj.vis = externalR THEN Dev2CPM.SymWInt(Srvar) ELSE Dev2CPM.SymWInt(Svar) END ;
                        OutStr(obj.typ); OutName(obj.name);
                        IF (obj.typ.strobj = NIL) OR (obj.typ.strobj.name = null) THEN
                            (* compute fingerprint to avoid structural type equivalence *)
                            Dev2CPM.FPrint(expCtxt.reffp, obj.typ.ref)
                        END
                    | XProc:
                        Dev2CPM.SymWInt(Sxpro); OutSign(obj.typ, obj.link); OutName(obj.name)
                    | IProc:
                        Dev2CPM.SymWInt(Sipro); OutSign(obj.typ, obj.link); OutName(obj.name)
                    | CProc:
                        Dev2CPM.SymWInt(Scpro); OutSign(obj.typ, obj.link); ext := obj.conval.ext;
                        IF ext # NIL THEN j := LEN(ext^); i := 0; Dev2CPM.SymWInt(j);
                            WHILE i < j DO Dev2CPM.SymWCh(ext[i]); INC(i) END
                        ELSE Dev2CPM.SymWInt(0)
                        END;
                        OutName(obj.name); portable := FALSE
                    END
                END
            END ;
            OutObj(obj.right)
        END
    END OutObj;

    PROCEDURE Export*(VAR ext, new: BOOLEAN);
            VAR i: INTEGER; nofmod: BYTE; done: BOOLEAN; old: Object; oldCSum: INTEGER;
    BEGIN
        symExtended := FALSE; symNew := FALSE; nofmod := nofGmod;
        Import("@self", SelfName, done); nofGmod := nofmod;
        oldCSum := Dev2CPM.checksum;
        ASSERT(GlbMod[0].name^ = SelfName);
        IF Dev2CPM.noerr THEN    (* ~Dev2CPM.noerr => ~done *)
            Dev2CPM.NewSym(SelfName);
            IF Dev2CPM.noerr THEN
                Dev2CPM.SymWInt(0);    (* portable symfile *)
                Dev2CPM.checksum := 0;    (* start checksum here to avoid problems with proc id fixup *)
                Dev2CPM.SymWInt(actVersion);
                old := GlbMod[0]; portable := TRUE;
                 IF libName # "" THEN
                    Dev2CPM.SymWInt(Slib); OutName(libName); portable := FALSE;
                    IF done & ((old.library = NIL) OR (old.library^ # libName)) THEN
                        FPrintErr(NIL, 252)
                    END
                ELSIF done & (old.library # NIL) THEN FPrintErr(NIL, 252)
                END;
                Dev2CPM.SymWInt(Smname); OutName(SelfName);
                expCtxt.reffp := 0; expCtxt.ref := FirstRef;
                expCtxt.nofm := 1; expCtxt.locmno[0] := 0;
                i := 1; WHILE i < maxImps DO expCtxt.locmno[i] := -1; INC(i) END ;
                OutObj(topScope.right);
                ext := sfpresent & symExtended;
                new := ~sfpresent OR symNew OR (Dev2CPM.checksum # oldCSum);
                IF Dev2CPM.noerr & ~portable THEN
                    Dev2CPM.SymReset;
                    Dev2CPM.SymWInt(processor)    (* nonportable symfile *)
                END;
                IF Dev2CPM.noerr & sfpresent & (impCtxt.reffp # expCtxt.reffp) THEN
                    new := TRUE
                END ;
                IF ~Dev2CPM.noerr THEN Dev2CPM.DeleteNewSym END
                (* Dev2CPM.RegisterNewSym is called in OP2 after writing the object file *)
            END
        END
    END Export;    (* no new symbol file if ~Dev2CPM.noerr *)


    PROCEDURE InitStruct(VAR typ: Struct; form: BYTE);
    BEGIN
        typ := NewStr(form, Basic); typ.ref := form; typ.size := 1; typ.allocated := TRUE;
        typ.strobj := NewObj(); typ.pbfp := form; typ.pvfp := form; typ.fpdone := TRUE;
        typ.idfp := form; typ.idfpdone := TRUE
    END InitStruct;

    PROCEDURE EnterBoolConst(IN name: Name; val: INTEGER);
        VAR obj: Object;
    BEGIN
        Insert(name, obj); obj.conval := NewConst();
        obj.mode := Con; obj.typ := booltyp; obj.conval.intval := val
    END EnterBoolConst;

    PROCEDURE EnterRealConst(IN name: Name; val: REAL; VAR obj: Object);
    BEGIN
        Insert(name, obj); obj.conval := NewConst();
        obj.mode := Con; obj.typ := real32typ; obj.conval.realval := val
    END EnterRealConst;

    PROCEDURE EnterTyp(IN name: Name; form: BYTE; size: SHORTINT; VAR res: Struct);
        VAR obj: Object; typ: Struct;
    BEGIN
        Insert(name, obj);
        typ := NewStr(form, Basic); obj.mode := Typ; obj.typ := typ; obj.vis := external;
        typ.strobj := obj; typ.size := size; typ.ref := form; typ.allocated := TRUE;
        typ.pbfp := form; typ.pvfp := form; typ.fpdone := TRUE;
        typ.idfp := form; typ.idfpdone := TRUE; res := typ
    END EnterTyp;
```

```
	PROCEDURE EnterProc(IN name: Name; num: SHORTINT);
		VAR obj: Object;
	BEGIN Insert(name, obj);
		obj.mode := SProc; obj.typ := notyp; obj.adr := num
	END EnterProc;

	PROCEDURE EnterAttr(IN name: Name; num: SHORTINT);
		VAR obj: Object;
	BEGIN Insert(name, obj);
		obj.mode := Attr; obj.adr := num
	END EnterAttr;

	PROCEDURE EnterTProc(ptr, rec: Struct; IN name: Name; num, typ: SHORTINT);
		VAR obj, par: Object;
	BEGIN
		InsertField(name, rec, obj);
		obj.mnolev := -128;	(* for correct implement only behaviour *)
		obj.mode := TProc; obj.num := num; obj.conval := NewConst();
		obj.conval.setval := obj.conval.setval + {newAttr};
		IF typ = 0 THEN	(* FINALIZE, RELEASE *)
			obj.typ := notyp; obj.vis := externalR;
			INCL(obj.conval.setval, empAttr)
		ELSIF typ = 1 THEN	(* QueryInterface *)
			par := NewObj(); par.name := NewName("int"); par.mode := VarPar; par.vis := outPar;
			par.sysflag := 8; par.adr := 16; par.typ := punktyp;
			par.link := obj.link; obj.link := par;
			par := NewObj(); par.name := NewName("iid"); par.mode := VarPar; par.vis := inPar;
			par.sysflag := 16; par.adr := 12; par.typ := guidtyp;
			par.link := obj.link; obj.link := par;
			obj.typ := restyp; obj.vis := external;
			INCL(obj.conval.setval, extAttr)
		ELSIF typ = 2 THEN	(* AddRef, Release *)
			obj.typ := notyp; obj.vis := externalR;
			INCL(obj.conval.setval, isHidden);
			INCL(obj.conval.setval, extAttr)
		END;
		par := NewObj(); par.name := NewName("this"); par.mode := Var;
		par.adr := 8; par.typ := ptr;
		par.link := obj.link; obj.link := par;
	END EnterTProc;

(*
	PROCEDURE EnterHdField(VAR root: Object; offs: SHORTINT);
		VAR obj: Object;
	BEGIN
		obj := NewObj(); obj.mode := Fld;
		obj.name := NewName(Dev2CPM.HdPtrName); obj.typ := undftyp; obj.adr := offs;
		obj.link := root; root := obj
	END EnterHdField;
*)

BEGIN
	NEW(null, 1); null^ := "";
	topScope := NIL; OpenScope(0, NIL); Dev2CPM.errpos := 0;
	InitStruct(undftyp, Undef); InitStruct(notyp, NoTyp);
	InitStruct(string8typ, String8); InitStruct(niltyp, NilTyp); niltyp.size := Dev2CPM.PointerSize;
	InitStruct(string16typ, String16);
	undftyp.BaseTyp := undftyp;

	(*initialization of module SYSTEM*)
```

```
(*
	EnterTyp("BYTE", Byte, 1, bytetyp);
	EnterProc("NEW", sysnewfn);
*)
	EnterTyp("PTR", Pointer, Dev2CPM.PointerSize, sysptrtyp);
	EnterProc("ADR", adrfn);
	EnterProc("TYP", typfn);
	EnterProc("CC", ccfn);
	EnterProc("LSH", lshfn);
	EnterProc("ROT", rotfn);
	EnterProc("GET", getfn);
	EnterProc("PUT", putfn);
	EnterProc("GETREG", getrfn);
	EnterProc("PUTREG", putrfn);
	EnterProc("BIT", bitfn);
	EnterProc("VAL", valfn);
	EnterProc("MOVE", movefn);
	EnterProc("THISRECORD", thisrecfn);
	EnterProc("THISARRAY", thisarrfn);
	syslink := topScope.right; topScope.right := NIL;

	(* initialization of module COM *)
	EnterProc("ID", iidfn);
	EnterProc("QUERY", queryfn);
	EnterTyp("RESULT", Int32, 4, restyp);
	restyp.ref := Res;
	EnterTyp("GUID", Guid, 16, guidtyp);
	guidtyp.form := Comp; guidtyp.comp := Array; guidtyp.n := 16;
	EnterTyp("IUnknown^", IUnk, 12, iunktyp);
	iunktyp.form := Comp; iunktyp.comp := Record; iunktyp.n := 3;
	iunktyp.attribute := absAttr;
(*
	EnterHdField(iunktyp.link, 12);
*)
	iunktyp.BaseTyp := NIL; iunktyp.align := 4;
	iunktyp.sysflag := interface; iunktyp.untagged := TRUE;
	NEW(iunktyp.ext, 40); iunktyp.ext^ := "{00000000-0000-0000-C000-000000000046}";
	EnterTyp("IUnknown", PUnk, Dev2CPM.PointerSize, punktyp);
	punktyp.form := Pointer; punktyp.BaseTyp := iunktyp;
	punktyp.sysflag := interface; punktyp.untagged := TRUE;
	EnterTProc(punktyp, iunktyp, "QueryInterface", 0, 1);
	EnterTProc(punktyp, iunktyp, "AddRef", 1, 2);
	EnterTProc(punktyp, iunktyp, "Release", 2, 2);
	comlink := topScope.right; topScope.right := NIL;

	universe := topScope;
	EnterProc("LCHR", lchrfn);
	EnterProc("LENTIER", lentierfcn);
	EnterTyp("ANYREC", AnyRec, 0, anytyp);
	anytyp.form := Comp; anytyp.comp := Record; anytyp.n := 1;
	anytyp.BaseTyp := NIL; anytyp.extlev := -1;	(* !!! *)
	anytyp.attribute := absAttr;
	EnterTyp("ANYPTR", AnyPtr, Dev2CPM.PointerSize, anyptrtyp);
	anyptrtyp.form := Pointer; anyptrtyp.BaseTyp := anytyp;
	EnterTProc(anyptrtyp, anytyp, "FINALIZE", 0, 0);
	EnterTProc(anyptrtyp, iunktyp, "RELEASE", 1, 0);
	EnterProc("VALID", validfn);

	EnterTyp("SHORTCHAR", Char8, 1, char8typ);
	string8typ.BaseTyp := char8typ;
```

```
    EnterTyp("CHAR", Char16, 2, char16typ);
    EnterTyp("LONGCHAR", Char16, 2, lchar16typ);
    string16typ.BaseTyp := char16typ;
    EnterTyp("SET", Set, 4, settyp);
    EnterTyp("BYTE", Int8, 1, int8typ);
    guidtyp.BaseTyp := int8typ;
    EnterTyp("SHORTINT", Int16, 2, int16typ);
    EnterTyp("INTEGER",  Int32, 4, int32typ);
    EnterTyp("LONGINT", Int64, 8, int64typ);
    EnterTyp("LARGEINT", Int64, 8, lint64typ);
    EnterTyp("SHORTREAL", Real32, 4, real32typ);
    EnterTyp("REAL", Real64, 8, real64typ);
    EnterTyp("LONGREAL", Real64, 8, lreal64typ);
    EnterTyp("BOOLEAN", Bool, 1, booltyp);
    EnterBoolConst("FALSE", 0);     (* 0 and 1 are compiler internal representation only *)
    EnterBoolConst("TRUE",  1);
    EnterRealConst("INF", Dev2CPM.InfReal, infinity);
    EnterProc("HALT", haltfn);
    EnterProc("NEW", newfn);
    EnterProc("ABS", absfn);
    EnterProc("CAP", capfn);
    EnterProc("ORD", ordfn);
    EnterProc("ENTIER", entierfn);
    EnterProc("ODD", oddfn);
    EnterProc("MIN", minfn);
    EnterProc("MAX", maxfn);
    EnterProc("CHR", chrfn);
    EnterProc("SHORT", shortfn);
    EnterProc("LONG", longfn);
    EnterProc("SIZE", sizefn);
    EnterProc("INC", incfn);
    EnterProc("DEC", decfn);
    EnterProc("INCL", inclfn);
    EnterProc("EXCL", exclfn);
    EnterProc("LEN", lenfn);
    EnterProc("COPY", copyfn);
    EnterProc("ASH", ashfn);
    EnterProc("ASSERT", assertfn);
(*
    EnterProc("ADR", adrfn);
    EnterProc("TYP", typfn);
*)
    EnterProc("BITS", bitsfn);
    EnterAttr("ABSTRACT", absAttr);
    EnterAttr("LIMITED", limAttr);
    EnterAttr("EMPTY", empAttr);
    EnterAttr("EXTENSIBLE", extAttr);
    NEW(intrealtyp); intrealtyp^ := real64typ^;
    impCtxt.ref[Undef] := undftyp; impCtxt.ref[Byte] := bytetyp;
    impCtxt.ref[Bool] := booltyp;  impCtxt.ref[Char8] := char8typ;
    impCtxt.ref[Int8] := int8typ;  impCtxt.ref[Int16] := int16typ;
    impCtxt.ref[Int32] := int32typ;  impCtxt.ref[Real32] := real32typ;
    impCtxt.ref[Real64] := real64typ;  impCtxt.ref[Set] := settyp;
    impCtxt.ref[String8] := string8typ; impCtxt.ref[NilTyp] := niltyp;
    impCtxt.ref[NoTyp] := notyp; impCtxt.ref[Pointer] := sysptrtyp;
    impCtxt.ref[AnyPtr] := anyptrtyp; impCtxt.ref[AnyRec] := anytyp;
    impCtxt.ref[Char16] := char16typ; impCtxt.ref[String16] := string16typ;
    impCtxt.ref[Int64] := int64typ;
    impCtxt.ref[IUnk] := iunktyp; impCtxt.ref[PUnk] := punktyp;
    impCtxt.ref[Guid] := guidtyp; impCtxt.ref[Res] := restyp;
```

END Dev2CPT.

```
Objects:

    mode   | adr     conval  link      scope     leaf
   ------------------------------------------------
    Undef  |                                          Not used
    Var    | vadr            next                regopt  Glob or loc var or proc value parameter
    VarPar | vadr            next                regopt  Var parameter (vis = 0 | inPar | outPar)
    Con    |         val                              Constant
    Fld    | off             next                     Record field
    Typ    |                                          Named type
    LProc  | entry   sizes   firstpar  scope     leaf     Local procedure, entry adr set in back-end
    XProc  | entry   sizes   firstpar  scope     leaf     External procedure, entry adr set in back-end
    SProc  | fno     sizes                            Standard procedure
    CProc  |         code    firstpar  scope          Code procedure
    IProc  | entry   sizes             scope     leaf     Interrupt procedure, entry adr set in back-end
    Mod    |                           scope          Module
    Head   | txtpos          owner     firstvar       Scope anchor
    TProc  | entry   sizes   firstpar  scope     leaf     Bound procedure, mthno = obj.num

    Structures:

    form      comp    | n       BaseTyp    link      mno   txtpos    sysflag
  ----------------------------------------------------------------------------------
-
    Undef     Basic   |
    Byte      Basic   |
    Bool      Basic   |
    Char8     Basic   |
    Int8      Basic   |
    Int16     Basic   |
    Int32     Basic   |
    Real32    Basic   |
    Real64    Basic   |
    Set       Basic   |
    String8   Basic   |
    NilTyp    Basic   |
    NoTyp     Basic   |
    Pointer   Basic   |         PBaseTyp             mno   txtpos    sysflag
    ProcTyp   Basic   |         ResTyp     params    mno   txtpos    sysflag
    Comp      Array   | nofel   ElemTyp              mno   txtpos    sysflag
    Comp      DynArr  | dim     ElemTyp              mno   txtpos    sysflag
    Comp      Record  | nofmth  RBaseTyp   fields    mno   txtpos    sysflag
    Char16    Basic   |
    String16  Basic   |
    Int64     Basic   |

Nodes:

design   = Nvar|Nvarpar|Nfield|Nderef|Nindex|Nguard|Neguard|Ntype|Nproc.
expr     = design|Nconst|Nupto|Nmop|Ndop|Ncall.
nextexpr = NIL|expr.
ifstat   = NIL|Nif.
casestat = Ncaselse.
sglcase  = NIL|Ncasedo.
stat     = NIL|Ninittd|Nenter|Nassign|Ncall|Nifelse|Ncase|Nwhile|Nrepeat|
           Nloop|Nexit|Nreturn|Nwith|Ntrap.


              class      subcl      obj       left       right      link
              ---------------------------------------------------------

design        Nvar                  var                             nextexpr
```

	Nvarpar		varpar			nextexpr	
	Nfield		field	design		nextexpr	
	Nderef	ptr/str		design		nextexpr	
	Nindex			design	expr	nextexpr	
	Nguard			design		nextexpr	(typ = guard type)
	Neguard			design		nextexpr	(typ = guard type)
	Ntype		type			nextexpr	
	Nproc	normal	proc			nextexpr	
		super	proc			nextexpr	
expr	design						
	Nconst		const				(val = node.conval)
	Nupto			expr	expr	nextexpr	
	Nmop	not		expr		nextexpr	
		minus		expr		nextexpr	
		is	tsttype	expr		nextexpr	
		conv		expr		nextexpr	
		abs		expr		nextexpr	
		cap		expr		nextexpr	
		odd		expr		nextexpr	
		bit		expr		nextexpr	{x}
		adr		expr		nextexpr	SYSTEM.ADR
		typ		expr		nextexpr	SYSTEM.TYP
		cc		Nconst		nextexpr	SYSTEM.CC
		val		expr		nextexpr	SYSTEM.VAL
	Ndop	times		expr	expr	nextexpr	
		slash		expr	expr	nextexpr	
		div		expr	expr	nextexpr	
		mod		expr	expr	nextexpr	
		and		expr	expr	nextexpr	
		plus		expr	expr	nextexpr	
		minus		expr	expr	nextexpr	
		or		expr	expr	nextexpr	
		eql		expr	expr	nextexpr	
		neq		expr	expr	nextexpr	
		lss		expr	expr	nextexpr	
		leq		expr	expr	nextexpr	
		grt		expr	expr	nextexpr	
		geq		expr	expr	nextexpr	
		in		expr	expr	nextexpr	
		ash		expr	expr	nextexpr	
		msk		expr	Nconst	nextexpr	
		len		design	Nconst	nextexpr	
		min		expr	expr	nextexpr	MIN
		max		expr	expr	nextexpr	MAX
		bit		expr	expr	nextexpr	SYSTEM.BIT
		lsh		expr	expr	nextexpr	SYSTEM.LSH
		rot		expr	expr	nextexpr	SYSTEM.ROT
	Ncall		fpar	design	nextexpr	nextexpr	
	Ncomp			stat	expr	nextexpr	
nextexpr	NIL						
	expr						
ifstat	NIL						
	Nif			expr	stat	ifstat	
casestat	Ncaselse			sglcase	stat		(minmax = node.conval)
sglcase	NIL						
	Ncasedo			Nconst	stat	sglcase	

stat	NIL						
	Ninittd					stat	(of node.typ)
	Nenter		proc	stat	stat	stat	(proc=NIL for mod)
	Nassign	assign		design	expr	stat	
		newfn		design	nextexp	stat	
		incfn		design	expr	stat	
		decfn		design	expr	stat	
		inclfn		design	expr	stat	
		exclfn		design	expr	stat	
		copyfn		design	expr	stat	
		getfn		design	expr	stat	SYSTEM.GET
		putfn		expr	expr	stat	SYSTEM.PUT
		getrfn		design	Nconst	stat	SYSTEM.GETREG
		putrfn		Nconst	expr	stat	SYSTEM.PUTREG
		sysnewfn		design	expr	stat	SYSTEM.NEW
		movefn		expr	expr	stat	SYSTEM.MOVE (right.link = 3rd par)
	Ncall		fpar	design	nextexpr	stat	
	Nifelse			ifstat	stat	stat	
	Ncase			expr	casestat	stat	
	Nwhile			expr	stat	stat	
	Nrepeat			stat	expr	stat	
	Nloop			stat		stat	
	Nexit					stat	
	Nreturn		proc	nextexpr		stat	(proc = NIL for mod)
	Nwith			ifstat	stat	stat	
	Ntrap				expr	stat	
	Ncomp			stat	stat	stat	