IDoc Outbound Triggers

IDocs should be sent out at certain events. Therefore you have to define a trigger. A lot of consideration is required to determine the correct moment when to send out the IDoc. The IDoc can be triggered at a certain time or when an event is raised. R/3 uses several completely different methods to determine the trigger point. There are messages to tell the system that there is an IDoc waiting for dispatching, there are log files which may be evaluated to see if IDocs are due to send and there can be a workflow chain triggered, which includes the sending of the IDoc.
9.1 Individual ABAP

The simplest way to create IDocs, is to write an ABAP which simply does it.

The individual ABAP can either be a triggering ABAP which runs at certain events, e.g. every night, or it can be an ABAP which does the complete IDoc creation from scratch.

Triggering ABAP

A triggering ABAP would simply try to determine which IDocs need sending and call the appropriate IDoc creation routines.

ABAP creates the whole IDoc

You may also imagine the ABAP to do all the job. As this is mostly reinventing the wheel, it is not really recommended and should be reserved to situation, where the other solution do not provide an appropriate mean.
9.2 NAST Messages Based Outbound IDocs

You can use the R/3 message concept to trigger IDocs the same way as you trigger SapScript printing.

One of the key tables in R/3 is the table NAST. This table records reminders written by applications. Those reminders are called messages.

Applications write messages to NAST, which will be processed by a message handler. Every time when an application sees the necessity to pass information to a third party, a message is written to NAST. A message handler will eventually check the entries in the table and cause an appropriate action.

The concept of NAST messages has originally been designed for triggering SapScript printing. The very same mechanism is used for IDocs, where the IDoc processor replaces the print task, as an IDoc is only the paperless form of a printed document.

EDI uses the same mechanism as printing. The messages are usually be created using the condition technique, a mechanism available to all major R/3 applications.

Condition technique can mostly be used. The messages are typically processed by FORM ENTRY in PROGRAM RSNAST00. The following piece of code does principally the same thing as RSNAST00 and makes full use of all customizing settings for message handling.

The conditions are set up the same way for any output media. So you may define a condition for printing a document and then just change the output media from printer to IDoc/EDI or ALE.

Printing, EDI and ALE use the same trigger. Programs RSNAST00 and RSNASTED provide versatile subroutines for NAST processing.

The concepts are mostly the same. Creating NAST messages is a standard functionality in most of the SAP core applications. Those applications - e.g. VA01 - perform calls to the central function module MESSAGING of group V61B. The function module uses customizing entries, mainly those of the tables T681* to T685*.

A NAST output message is stored as a single record in the table NAST. The record stores all information that is necessary to create an IDoc. This includes mainly an object key to identify the processed object and application to the message handler and the sender and receiver information.

Programs RSNAST00 and RSNASTED provide FOR NAST processing.

The following piece of code does principally the same thing as RSNAST00 and makes full use of all customizing settings for message handling.

The processing routine for the respective media and message is customized in the table TNAPR. This table records the name of a FORM routine, which processes the message for the chosen media and the name of an ABAP where this FORM is found.

Illustration 10: Communicating with message via table NAST
9.3 The RSNAST00 ABAP

The ABAP RSNAST00 is the standard ABAP, which is used to collect unprocessed NAST messages and to execute the assigned action.

RSNAST00 is the standard batch collector for messages

RSNAST00 can be executed as a collector batch run, that eventually looks for unprocessed IDocs. The usual way of doing that is to define a batch-run job with transaction [SM37]. This job has to be set for periodic processing and start a program that triggers the IDoc re-sending.

RSNAST00 processes only messages of a certain status

Cave! RSNAST00 will only look for IDocs which are set to NAST-VSZTP = '1' or '2' (Time of processing). VSZPT = '3' or '4' is ignored by RSNAST00.

For batch execution a selection variant is required

Start RSNAST00 in the foreground first and find the parameters that match your required selection criteria. Save them as a VARIANT and then define the periodic batch job using the variant.

If RSNAST00 does not meet 100% your needs you can create an own program similar to RSNAST00. The only requirement for this program are two steps:

- Read the NAST entry to process into structure NAST

  TABLES NAST.
  DATA: subrc like sy-subrc.
  SELECT FROM NAST WHERE ......
  * then call FORM einzelnachricht(RSNAST00) to process the record
  PERFORM einzelnachricht(RSNAST00) USING subrc.

9.4 Sending IDocs Via RSNASTED

Standard R/3 provides you with powerful routines, to trigger, prepare and send out IDocs in a controlled way. There are only a few rare cases, where you do not want to send IDocs the standard way.

The ABAP RSNAST00 is the standard routine to send IDocs from entries in the message control. This program can be called directly, from a batch routine with variant or you can call the FORM einzelnachricht_screen(RSNAST00) from any other program, while having the structure NAST correctly filled with all necessary information.

If there is an entry in table NAST, RSNAST00 looks up the associated processing routine in table TNAPR. If it is to send an IDoc with standard means, this will usually be the routine RSNASTED(EDI_PROCESSING) or RSNASTED(ALE_PROCESSING) in the case of ALE distribution.

RSNASTED itself determines the associated IDoc outbound function module, executes it to fill the EDIDx tables and passes the prepared IDoc to the port.

You can call the standard processing routines from any ABAP, by executing the following call to the routine. You only have to make sure that the structure NAST is declared with the tables statement in the calling routine and that you fill at least the key part and the routine (TNAPR) information before.

TABLES NAST.
NAST-MANDT = SY-MANDT.
NAST-KSCHL = 'ZEDIK'.
NAST-KAPP = 'V1'.
NAST-OBJNKY = '0012345678'.
NAST-PARNR = 'D012345678'.
PERFORM einzelnachricht_screen(RSNAST00).

Calling einzelnachricht_screen determines how the message is processed. If you want to force the IDoc-processing you can call it directly:

TNAPR-PROGN = '0'.
TNAPR-ROUTN = 'ENTRY'.
PERFORM edi_processing(RSNASTED).
9.5 Sending IDocs Via RSNAST00

Here is the principle flow how RSNAST00 processes messages for IDocs.

REPORT rsnast00

CALL FUNCTION 'RSNAST00' EXPORTING

SELECT * FROM table

... PERFORM einzel_nachricht IN PROGRAM rsnast00

FORM einzelnachricht

SELECT * FROM table

... PERFORM einzel_nachricht IN PROGRAM rsnast00

9.6 Workflow Based Outbound IDocs

Unfortunately, there are application that do not create messages. This is especially true for master data applications. However, most applications fire a workflow event during update, which can easily be used to trigger the IDoc distribution.

SWE_EVENT_CREATE

Workflow is a call to a function module

Applications with change documents always trigger workflow events.

If an application writes regular change documents (ger.: Änderungsbelege) to the database, it will issue automatically a workflow event. This event is triggered from within the function CHANGEDOCUMENT_CLOSE. The change document workflow event is always triggered, independent of the case whether a change document is actually written.

In order to make use of the workflow for IDoc processing, you do not have to go through the cumbersome workflow design procedure as it is described in the workflow documentation. For the mentioned purpose, you can register the workflow handler from the menu, which says Event Coupling from the BALD transaction.

Workflow cannot easily be restarted

Triggering the IDoc from a workflow event has a disadvantage: if the IDoc has to be repeated for some reason, the event cannot be repeated easily. This is due to the nature of a workflow event, which is triggered usually from a precedent action. Therefore you have to find an own way how to make sure that the IDoc is actually generated, even in the case of an error. Practically this is not a very big problem for IDocs. In most cases the creation of the IDoc will always take place. If there is a problem, then the IDoc would be stored in the IDoc base with a respective status. It will shown in transaction WE05 and can be resend from there.
9.7 Workflow Event From Change Document

Instead of waiting for a polling job to create IDocs, they can also be created immediately after a transaction finishes. This can be done by assigning an action to a workflow event.

**Workflow events are usually fired from an update routine**

Most applications fire a workflow event from the update routine by calling the function:

```
FUNCTION swe_event_create
```

You can check if an application fires events by activating the event log from transaction SWLD.

**SWLD lets install and log workflows**

Calling and saving a transaction will write the event’s name and circumstances into the log file.

If an application does not fire workflow events directly, there is still another chance that a workflow may be used without touching the R/3 original programs. SWLD lets install and log workflows.

**Workflow events are also fired from change document**

Every application that writes change documents triggers a workflow event from within the function module CHANGEDOCUMENT_CLOSE, which is called from the update processing upon writing the change document. This will call the workflow processor:

```
FUNCTION swe_event_create_changedocument
```

Both workflow types are not compatible with each other with respect to the function modules used to handle the event.

The workflow types are incompatible but work according the same principal:

Both will call a function module whose name they find in the workflow linkage tables:

- `swecdobj-objtypefb` for the workflow event created by the update routine.
- `swecdobj-objtypefb` for the workflow event created by the update routine.

If a name is found, the function module will then be called dynamically. This is all to say about the linkage of the workflow.

The dynamic call looks like the following:

```
CALL FUNCTION swecdobj-objtypefb
EXPORTING
  changedocument_header = changedocument_header
  obj ect type = swecdobj-obj type
IMPORTING
  obj ect type = swecdobj-obj type
TABLES
  changedocument_position = changedocument_position.
```

9.8 ALE Change Pointers

Applications which write change documents will also try to write change pointers for ALE operations. These are log entries to remember all modified data records relevant for ALE.

**Change docs remember changes in transaction**

Change documents remember the modified fields made to the database by an application. They also remember the user name and the time when the modification took place.

**Data elements are marked to be relevant for change documents**

The decision whether a field modification is relevant for a change document is triggered by a flag of the modified field's data element. You can set the flag with `SE11` by modifying the data element.

**ALE may need other triggers**

For the purpose of distributing data via ALE to other systems, you may want to choose other fields, which shall be regarded relevant for triggering a distribution. Therefore R/3 introduced the concept of change pointers, which are nothing else than a second log file specially designed for writing the change pointers which are meant to trigger IDoc distribution via ALE.

**Change pointers remember key of the document**

So the change pointers will remember the key of the document every time when a relevant field has changed.

**An ABAP creates the IDocs**

Change pointers are then evaluated by an ABAP which calls the IDoc creation, for every modified document found in the change pointers.

**Change pointers are when change documents have been written**

The Change pointers are written from the routine CHANGEDOCUMENT_Close when saving the generated change document. So change pointers are automatically written when a relevant document changes.

The following function is called from within CHANGEDOCUMENT_CLOSE in order to write the change pointers:

```
CALL FUNCTION 'CHANGE_POINTERS_CREATE'
EXPORTING
  change_document_header = cdhdr
TABLES
  change_document_position = ins_cdpos.
```
9.9 Activation of change pointer update

Change pointers are log entries to table BDCP which are written every time a transaction modifies certain fields. The change pointers are designed for ALE distribution and written by the function CHANGE_DOCUMENT_CLOSE.

Change pointers are written for use with ALE. There are ABAPs like RBDMIDOC which can read the change pointers and trigger an IDoc for ALE distribution.

The change pointers are mainly the same as change documents. They however can be set up differently, so fields which trigger change documents are not necessarily the same that cause change pointers to be written.

In order to work with change pointers there are two steps to be performed:
1. Turn on change pointer update generally
2. Decide which message types shall be included for change pointer update

Activate Change Pointer Generally

R3 allows to activate or deactivate the change pointer update. For this purpose it maintains a table TBDME. The decision whether the change pointer update is active is done with a function module called 'Ale_Component_Check'.

Currently (release 40B) this check does nothing else than to check, if this table has an entry or not. If there is an entry in TBDME, the ALE change pointers are generally active. If this table is empty, change pointers are turned off for everybody and everything, regardless of the other settings.

The two points read like you had the choice between turning it on generally or selectively. This is not the case: you always turn them on selectively. The switch to turn on generally is meant to activate or deactivate the whole mechanism.

The change pointers which have not been processed yet, can be read with a function module.

Use Change Documents Instead Of Change Pointers

When you want to send out an IDoc unconditionally every time a transaction updates, you might want to use the workflow from change documents. However, for a more flexible approach, change pointers can be used.

9.10 Dispatching ALE IDocs for Change Pointers

Change pointers must be processed by an ABAP, e.g. RBDMIDOC.

RBDMIDOC processes change pointers and sends the IDocs

The actual distribution of documents from change pointers must be done by an ABAP, which reads the change pointers and processes them. The standard ABAP for that is RBDMIDOC. For recurring execution it can be submitted in a scheduled job using SM37.

Function module defined in table TBDME

It then calls dynamically a function module whose name is stored in table TBDME for each message type.

Example

A complex example for a function module, which collects the change pointers, can be examined in: MASTERIDOC_CREATE_SMD_DEBMAS. This one reads change pointers for debtors (customer masters). During the processing, it calls the actual IDoc creating module MASTERIDOC_CREATE_DEBMAS.

To summarize the change pointer concept

- Change pointers record relevant updates of transaction data
- Change pointers are written separate from the change documents, while at the same time
- Change pointers are evaluated by a collector run...
Dispatching ALE IDocs for Change Pointers

IDoc Outbound Triggers

### Chap 9

#### Illustration 12: Tables involved in change pointers processing

<table>
<thead>
<tr>
<th>Object</th>
<th>Table name</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBI</td>
<td>KNA1</td>
<td>NAME3</td>
</tr>
<tr>
<td>DEBI</td>
<td>Kann1</td>
<td>ORT01</td>
</tr>
<tr>
<td>DEBI</td>
<td>Kann1</td>
<td>REGIO</td>
</tr>
</tbody>
</table>

#### Illustration 13: Sample content of view V_TBD62

- **Object** | **Table name** | **Field** |
- DEBI       | KNA1         | NAME3    |
- DEBI       | Kann1        | ORT01    |
- DEBI       | Kann1        | REGIO    |

---

#### IDoc Recipes

The chapter shall show you how an IDoc function is principally designed and how R/3 processes the IDocs.
10.1 How the IDoc Engine Works

IDocs are usually created in a four step process. These steps are: retrieving the data, converting them to IDoc format, add a control record and delivering the IDoc to a port.

Collect data from R/3 database
This is the most individual task in outbound processing. You have to identify the database tables and data dependencies, which are needed in the IDoc to be sent. The smartest way is usually to select the data from database into an internal table using `SELECT * FROM dbtable INTO itab ... WHERE ...`.

Wrap data in IDoc format
The collected data must be transformed into ASCII data and filled into the predefined IDoc segment structures. The segment definitions are done with transaction WE31 and the segments allowed in an IDoc type are set up in transaction WE30. Segment once defined with WE31 are automatically created as SAP DDIC structures. They can be viewed with SE11, however they cannot be edited.

Create the IDoc control record
Every IDoc must be accompanied by a control record. This record must contain at least the IDoc type to identify the syntactical structure of the data and it must contain the name and role of the sender and the receiver. This header information is checked against the partner definitions for outbound. Only if a matching partner definition exists, the IDoc can be sent. Partner definitions are set up with transaction WE20.

Send data to port
When the partner profile check passes, the IDoc is forwarded to a logical port, which is also assigned in the partner profile. This port is set up with transaction WE21 and defines the medium to transport the IDoc, e.g. file or RFC. The RFC destinations are set up with transaction SM57 and must also be entered in table TBDLS with an SM31 view. Directories for outbound locations of files are set up with transaction FILE and directly in WE21. It also allows to use a function module which generate file names. Standard functions for that purpose begin like EDI_FILE*.

10.2 How SAP Standard Processes Inbound IDocs

When you receive an IDoc the standard way, the data is stored in the IDoc base and a function module is called, which decides how to process the received information.

EDID4 - Data
Data is stored in table EDID4 (EDID3 up to release 3.xx, EDIDD up to release 2.xx)

EDIDC - Control Record
An accompanying control record with important context and administrative information is stored in table EDIDC.

Event signals readiness
After the data is stored in the IDoc base tables, an event is fired to signal that there is an IDoc waiting for processing. This event is consumed by the IDoc handler, which decides, whether to process the IDoc immediately, postpone processing or decline activity for whatever reason.

EDIFCT - Processing function
When the IDoc processor thinks it is time to process the IDoc it will have a look into table EDIFCT, where it should find the name of a function module, which will be called to process the IDoc data.

This function module is the heart of all inbound processing. The IDoc processor will call this routine and pass the IDoc data from EDID4 and the control record from EDIDC for the respective IDoc.

Function has a fixed interface
Because this routine is called dynamically it must adhere to some conventions, where the most important ones are: the interface parameters of the function must match the following call:

EDIDS - Status log
The processing steps and their respective status results are stored in table EDIDS.

Status must be logged properly
In addition the routine has to determine properly the next status of the IDoc in table EDIDS, usually it will be EDIDS-STATU = 53 for OK or 51 for error.
10.3 How To Create the IDoc Data

R/3 provides a sophisticated IDoc processing framework. This framework determines a function module, which is responsible for creating or processing the IDoc.

Function Module to generate the IDoc

The kernel of the IDoc processing is always a distinct function module. For the outbound processing the function module creates the IDoc and leaves it in an internal table, which is passed as interface parameter. During inbound processing the function module receives the IDoc via an interface parameter table. It would interpret the IDoc data and typically update the database either directly or via a call transaction.

Function modules are called dynamically

The function modules are called dynamically from a standard routine. Therefore the function must adhere to a well defined interface.

Function group EDIN with useful routines

You may want to investigate the function group EDIN, which contains a number of IDoc handler routines and would call the customized function.

Copy and modify existing routines

The easiest way, to start the development of an Outbound IDoc function module, is to copy an existing one. There are many samples in the standard R/3 repository, most are named IDOC_OUTBOUND* or IDOC_OUTPUT*. Outbound sample functions are named like FUNCTION IDOC_OUTPUT_ORDERS01

Inbound sample functions are named like FUNCTION IDOC_INPUT_ORDERS01

Outbound sample functions for master data are named like FUNCTION MASTERIDOC_CREATE_MTMAS

Illustration 14: Schematic of an IDoc Outbound Process
10.4 Interface Structure of IDoc Processing Functions

To use the standard IDoc processing mechanism, the processing function module must have certain interface parameters, because the function is called dynamically from a standard routine.

The automated IDoc processor will call your function module from within the program RSNASTED, usually either from the FORM ALE_PROCESSING or EDI_PROCESSING.

In order to be compatible with this automated call, the interface of the function module must be compliant.

FUNCTION Z_IDOC_OUTBOUND_SAMPLE.

* IMPORTING
  VALUE(FL_TEST) LIKE RS38L-OPTONAL DEFAULT 'X'
  VALUE(FL_COMMIT) LIKE RS38L-OPTONAL DEFAULT SPACE

* EXPORTING
  VALUE(F_IDOC_HEADER) LIKE EDIDC STRUCTURE EDIDC
  TABLES
    T_IDOC_CONTRL STRUCTURE EDIDC
    T_IDOC_DATA STRUCTURE EDIDD

* CHANGING
  VALUE(CONTROL_RECORD_IN) LIKE EDIDC STRUCTURE EDIDC
  VALUE(OBJECT) LIKE NAST STRUCTURE NAST

* EXCEPTIONS
  ERROR_IN_IDOC_CONTROL
  ERROR_WRITING_IDOC_STATUS
  ERROR_IN_IDOC_DATA
  SENDING_LOGICAL.SYSTEM_UNKNOWN
  UNKNOWN_ERROR

Program 3: Interface structure of an NAST compatible function module

Inbound functions are also called via a standard mechanism.

FUNCTION Y_AXX_COOKBOOK_TEXT_IDOC_OUTB.

* IMPORTING
  VALUE(I_TDOBJECT) LIKE THEAD-TDOBJECT DEFAULT 'TEXT'
  VALUE(I_TDID) LIKE THEAD-TDID DEFAULT 'ST'
  VALUE(I_TDNAME) LIKE THEAD-TDNAME
  VALUE(I_TDSPRAS) LIKE THEAD-TDSPRAS DEFAULT SY-LANGU

* EXPORTING
  VALUE(E_THEAD) LIKE THEAD STRUCTURE THEAD
  TABLES
    IDOC_CONTRL STRUCTURE EDIDC
    IDOC_DATA STRUCTURE EDIDD
    IDOC_STATUS STRUCTURE BDIDOCSTAT
    RETURN_VARIABLES STRUCTURE BDWFRETVAR
    SERIALIZATION_INFO STRUCTURE BDII SER

* PERFORM stuff the data into the IDoc record format
  CALL FUNCTION 'READ_TEXT'
  EXPORTING
    ID = ID
    LANGUAGE = LANGUAGE
    NAME = NAME
    OBJECT = OBJECT
    TABLES
    LI NES = LI NES.

Program 4: Interface structure of an IDoc inbound function

10.5 Recipe To Develop An Outbound IDoc Function

This is an individual coding part where you need to retrieve the information from the database and prepare it in the form the recipient of the IDoc will expect the data.

Read data to send

The first step is reading the data from the database, the one you want to send.

FUNCTION Y_AXX_COOKBOOK_TEXT_IDOC_OUTB.

* IMPORTING
  VALUE(INPUT_METHOD) LIKE BDWFAP_PAR-INPUTMETHD
  VALUE(MASS_PROCESSING) LIKE BDWFAP_PAR-MASS_PROC

* EXPORTING
  VALUE(WORKFLOW_RESULT) LIKE BDWFAP_PAR-RESULT
  VALUE(APPLICATION_VARIABLE) LIKE BDWFAP_PAR-APPL_VAR
  VALUE(IN_UPDATE_TASK) LIKE BDWFAP_PAR-UPDATE_TASK
  VALUE(CALL_TRANSACTION_DONE) LIKE BDWFAP_PAR-CALLTRANS

* TABLES
  IDOC_CONTRL STRUCTURE EDIDC
  IDOC_DATA STRUCTURE EDIDD
  IDOC_STATUS STRUCTURE BDIDOCSTAT
  RETURN_VARIABLES STRUCTURE BDWFRETVAR
  SERIALIZATION_INFO STRUCTURE BDII SER

* PERFORMANCE PACK the data into the IDoc record format
  CALL FUNCTION 'PACK_LINES'
  EXPORTING
    THEAD = THEAD
    LANGUAGE = LANGUAGE
    NAME = NAME
    OBJECT = OBJECT
    TABLES
    LI NES = LI NES.

Program 4: Interface structure of an IDoc inbound function
10.6 Converting Data Into IDoc Segment Format

The physical format of the IDocs records is always the same. Therefore the application data must be converted into a 1000 character string.

Fill the data segments which make up the IDoc

An IDoc file has a rigid formal structure. This allows the correspondents to correctly interpret the IDoc information. Were it for data exchange between SAP-systems only, the IDoc segments could be simply structured like the correspondent DDIC structure of the tables whose data is sent.

However, IDocs are usually transported to a variety of legacy systems which do not run SAP. Both correspondents therefore would agree an IDoc structure which is known to the sending and the receiving processes.

All data needs to be compiled in an internal table with the structure of the standard SAP table EDIDD. The records for EDIDD are principally made up of a header string describing the segment and a variable length character field (called SDATA) which will contain the actual segment data.

Program 5: Routine to move the translate to IDoc data

```as
FORM PACK_LINE TABLES IDOC_DATA USING 'THEAD' E_THEAD.
TABLES: THEAD.
MOVE-CORRESPONDING E:THEAD to Z1THEAD.
MOVE 'Z1THEAD' TO IDOC_DATA-SEGNAM.
MOVE Z1THEAD TO IDOC_DATA-SDATA.
APPEND IDOC_DATA.
ENDFORM.
```

Program 6: Fill the essential information of an IDoc control record

Finally the control record has to be filled with meaningful data, especially telling the IDoc type and message type.

```
IF IDOC_CONTRL-SNDPRN IS INITIAL.
   SELECT SINGLE * FROM T000 WHERE MANDT EQ SY-MANDT.
   MOVE T000-LOGSYS TO IDOC_CONTRL-SNDPRN.
ENDIF.
IDOC_CONTRL-SNDPRT = 'LS'.
IDOC_CONTRL-OUTMOD = '2'.
CLEAR IDOC_CONTRL.
IDOC_CONTRL-IDOCTP = 'YAXX_TEXT'.
APPEND IDOC_CONTRL.
```

Partner Profiles and Ports

R/3 defines partner profiles for every EDI partner. The profiles are used to declare the communication channels, schedule and conditions of processing.

Summary

- Partner profiles declare the communication medium to be used with a partner
- Ports define the physical characteristics of a communication channel
- If you define an ALE scenario for your IDoc partners, you can use the ALE automated partner profile generation (→ ALE)
11.1 IDoc Type and Message Type

An IDoc file requires a minimum of accompanying information to give sense to it. These are the message type and the IDoc type. While the IDoc type tells you about the fields and segments of the IDoc file, the message type flags the context under which the IDoc was sent.

IDoc Type signals Syntactical Structure

A receiver of an IDoc must exactly know the syntactical structure of the data package received. Naturally, the receiver only sees a text file with lines of characters. In order to interpret it, it is necessary to know, which segment types the file may contain and how a segment is structured into fields. SAP sends the name of the IDoc type in the communication header.

IDoc type (WE30)

The IDoc type describes the file structure. The IDoc type is defined and viewable with transaction WE30.

Examples:

- Examples of IDoc types are: MATMAS01, ORDERS01, COND_A01, or CLSMAS01.

Message Type signals the semantic context

The message type is an identifier that tags the IDoc to tell the receiver, how the IDoc is meant to be interpreted. It is therefore the tag for the semantic content of the IDoc.

Examples:

- Examples of IDoc types are: MATMAS, ORDERS, COND_A, or CLSMAS.

For any combination of message type and receiving partner, a profile is maintained

The combination of IDoc type and message type gives the IDoc the full meaning. Theoretically you could define only a single IDoc type for every IDoc you send. Then, all IDocs would have the same segments and the segments would have always the same field structure. According to the context some of the record fields are filled, others are simply void. Many ancient interfaces are still working that way.

Typical combinations of IDoc and message types are the following:

<table>
<thead>
<tr>
<th>Message Type</th>
<th>IDoc Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sales order, older format</td>
<td>ORDERS ORDERS01</td>
</tr>
<tr>
<td>Sales order, newer format</td>
<td>ORDERS ORDERS02</td>
</tr>
<tr>
<td>Purchase Requisition</td>
<td>PURREQ ORDERS01</td>
</tr>
</tbody>
</table>

The example shows you, that sales orders can be exchanged in different file formats. There may be some customers who accept the latest IDoc format (ORDERS02), while others still insist in receiving the old format (ORDERS01).

The IDoc format for sales orders would also be used to transfer a purchase requisition. While the format remains the same, the different message type signals, that this is not an actual order but a request.

11.2 Partner Profiles

Partner profiles play an important role in EDI communications. They are parameter files which store the EDI partner dependent information.

When data is exchanged between partners it is important that sender and receiver agree about the exact syntax and semantics of the data to be exchanged. This agreement is called a partner profile and tells the receiver the structure of the sent file and how its content is to be interpreted.

For any combination of message type and receiving partner, a profile is maintained

The information defined with the partner profile are:

- Names of sender and receiver to exchange the IDoc information for the respective IDoc and message type
- Logical port name via which the sender and receiver, resp. will communicate
- The communication media is assigned by the profile

If you exchange e.g. sales orders with partners, you may do this via different media with different customers. There may be one customer to communicate with you via TCP/IP (the Internet) while the other still insists in receiving diskette files.

Profiles cannot be transported

They must be defined for every R/3 client individually. They cannot be transported using the R/3 transport management system. This is because the profile contain the name of the sending system, which are naturally different for consolidation and production systems.

Profiles define the allowed EDI connections

The profiles allow you to open and close EDI connection with individual partners and specify in detail which IDocs are to be exchanged via the interface.

Profiles can also used to block an EDI communication

The profile is also the place to lock permanently or temporarily an IDoc communication with an EDI partner. So you shut the gate for external communication with the profile.
11.3 Defining the partner profile (WE20)

The transaction WE20 is used to set up the partner profile.

**WE20**

The profiles are defined with transaction **WE20** which is also found in the EDI master menu. From there you need to specify partner and partner type and whether you define a profile for inbound or outbound. Additionally you may assign the profile to a NAST message type.

**Partner type, e.g.**
- **LI=Supplier**
- **CU=Customer**
- **LS=Logical system**

**Inbound and Outbound definitions**

For every partner and every direction of communication, whether you receive or send IDocs, a different profile is maintained. The inbound profile defines the processing routine. The outbound profile defines the target, where to send the data to.

**Link message type to outbound profile**

If you send IDocs out of an application’s messaging, i.e. a communication via the NAST table, then you have to link the message type with an IDoc profile. This is also done in transaction **WE20.**

**Inbound profiles determine the processing logic**

The processing code is a logical name for the processing function module or object method. The processing code is used to uniquely determine a function module that will process the received IDoc data. The inbound profile will point to a processing code.

11.4 Data Ports (WE21)

IDoc data can be sent and received through a multitude of different media. In order to decouple the definition of the media characteristics from the application using it, the media is accessed via ports.

**A port is a logical name to access a physical input/output device**

A port is a logical name for an input/output device. A program talks to a port which is presented to it with a common standard interface. The port takes care of the translation between the standard interface format and the device dependent format.

**Communication media is defined via a port definition**

Instead of defining the communication path directly in the partner profile, a port number is assigned rather. The port number then designates the actual medium. This allows to define the characteristics of a port individually and use that port in multiple profiles. Changes in the port will then reflect automatically to all profiles without touching them.

**Typical ports for data exchange are:**
- Disk file with a fixed name
- Disk file with dynamic names
- Disk file with trigger of a batch routine
- Standard RFC connection via TCP/IP
- A network channel
- TCP/IP FTP destination (The Internet)
- Call to a individual program e.g. EDI converter

**Every program should communicate with other computers via the ports only**

Every application should send or receive its data via the logical ports only. This allows to easily change the hardware and software used to make the physical I/O connection without interfering with the programs itself.

The transactions used to define the ports are

**WE21 defines the port; SM59 sets up media**

To create the port and assign a logical name to define the physical characteristics of the I/O device used.

There are different port versions for the respective R/3 releases as shown in the matrix below:

<table>
<thead>
<tr>
<th>Port Type</th>
<th>DDic Format</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>not used</td>
<td>not used</td>
</tr>
<tr>
<td>2</td>
<td>EDID3</td>
<td>2.x, 3.x</td>
</tr>
<tr>
<td>3</td>
<td>EDID4</td>
<td>4.x</td>
</tr>
</tbody>
</table>

**Illustration 15: R/3 port types by release**

**Port versions differ in length of fields**

The difference between the port types are mainly the length of some fields. E.g. does port type 3 allow segment names up to 30 characters in length, while port type 3 is constraint to a maximum segment name of 8 characters.